SICHUAN UNIVERSITY

本科生毕业设计(学术论文)

Undergraduate Graduation Project

(Academic Thesis)

Title **LLM Fine-Tune For Medical Student Assistance: MedPredict**

School **College of Software Engineering**

Major **Software Engineering**

Student's Name **Filali Salma**

Student ID: **2021521460212** Grade **2021**

Adviser **Tang Mingjie**

教务处制表

2025 年 **5** 月 **20** 日

Made by the Office of Academic Affairs

May 20, 2025

ABSTRACT

Software Engineering

Student: Filali Salma Adviser: Tang Mingjie

【Abstract】This document outlines the full-stack development and implementation of MedPredict, an interactive, multi-functional web platform designed to assist medical students in exam preparation through AI-driven tools. At its core, MedPredict integrates a fine-tuned transformer-based language model, PubMedBERT, adapted specifically for medical multiple-choice question answering (MCQA). Built for practical use, the platform includes key features such as a flashcard generator for active recall of complex definitions, a clinical case simulator for real-life diagnostic practice, and a symptom-based disease predictor. The most advanced component is an LLM-powered MCQA module that automatically identifies the most probable correct answer from four options.

MedPredict was developed using modern full-stack technologies, combining natural language processing (NLP) and scalable web frameworks to deliver a seamless and responsive user experience. The platform is optimized for medical students preparing for high-stakes exams such as AIIMS, NEET-PG, and USMLE, offering practical tools aligned with real exam formats. The MCQA system was fine-tuned using the MedMCQA dataset, which includes over 194,000 questions across 21 medical subjects, ensuring relevance to postgraduate-level content. Training involved standard supervised learning over three epochs, with extensive preprocessing to ensure clean, usable data for deployment.

This project demonstrates how advanced biomedical NLP models can be deployed in functional, user-oriented applications. MedPredict serves as a working example of AI-assisted learning tools, with future-ready architecture that allows for upcoming features such as multilingual support, educator dashboards, GPT-based explanations, and richer clinical simulation. The development of MedPredict prioritizes usability, educational value, and privacy, marking a significant contribution to the growing landscape of AI-driven educational platforms in

healthcare.

【**Keywords**】**:** Biomedical Question Answering, Educational Web Application, Vue.js, Node.js, PubMedBERT Fine-Tuning.

摘 要

软件工程

学生 Filali Salma 指导教师: 唐明杰

[摘要]本文全面介绍了MedPredict的全栈开发与实现过程。MedPredict 是一个交互式、多功能的网页平台，旨在通过人工智能驱动的工具帮助医学生备考。该平台的核心是一个经过精调的基于 Transformer 架构的语言模型 PubMedBERT，专门适配于医学多项选择题（MCQA）的自动答题任务。

MedPredict 集成了多个实用功能，包括用于记忆复杂医学定义的抽认卡生成器、模拟真实诊疗情景的临床病例模拟器，以及一个基于症状的疾病预测工具。其中最先进的功能是一个由大语言模型驱动的 MCQA 模块，能够在四个选项中自动判断最可能的正确答案。

该平台采用现代全栈技术开发，将自然语言处理（NLP）与可扩展的网页框架结合，提供流畅且响应迅速的用户体验。MedPredict 专为备考高难度医学考试的学生设计，如AIIMS、NEET-PG 以及国际考试如 USMLE，其功能严格对标实际考题格式。MCQA 模型使用包含超过 194,000 道题目的 MedMCQA 数据集进行精调，涵盖 21 个临床与基础学科，确保内容贴合研究生级别的考试要求。训练过程采用标准监督学习，共进行三轮训练，并对数据进行了格式化、分词与去噪等预处理操作，确保模型部署质量。

本项目展示了先进的生物医学 NLP 模型如何在实际可用的应用中得到部署。MedPredict 是一个真实可用的 AI 助学工具范例，其架构支持未来扩展功能，包括多语言支持、教师端控制面板、基于 GPT 的题目讲解模块，以及更复杂的临床病例整合。MedPredict 的开发注重可用性、教育价值与用户隐私，为 AI 在医疗教育中的应用发展作出了积极贡献.

【关键词】生物医学问答，教育 Web 应用，Vue.js，Node.js，PubMedBERT 微调

Table of Contents

List of Figures

List of tables

List of Listings

List of abbreviations

API: Application Programming Interface

BERT: Bidirectional Encoder Representations from Transformers

CSV: Comma-Separated Values

EHR: Electronic Health Record

GPU: Graphics Processing Unit

JSON: JavaScript Object Notation

LLM: Large Language Model

MCQ: Multiple-Choice Question

MCQA: Multiple-Choice Question Answering

NER: Named Entity Recognition

NLP: Natural Language Processing

PDF: Portable Document Format

REST: Representational State Transfer

RAG: Retrieval-Augmented Generation

UI: User Interface

UML: Unified Modeling Language

USMLE: United States Medical Licensing Examination

Chapter 1 Introduction

1.1 Background and Motivation

1.1.1 Background

In recent years, natural language processing has progressed from academic theory to real-world deployment, largely thanks to the rise of transformer-based large language models. Models like BERT, GPT, and their domain-specialized versions have not only set new benchmarks across NLP tasks but have also opened the door to building practical tools for everyday use. In this project, I focus on bringing those capabilities into a usable system—specifically in the domain of medical education—where fast, reliable, and interactive learning tools are in high demand. Instead of treating these models as abstract research objects, I use them as core components in building a platform that solves actual learning challenges faced by medical students.

In highly technical domains like biomedicine, where the stakes are high and the information load is dense, tools that combine domain-specific language models with structured educational interfaces can offer meaningful support. AI in healthcare is not new, but its application in learning environments is still in the early stages. The complexity of medical knowledge—combined with the time pressure faced by students—makes this an ideal area for intelligent tutoring systems. My goal with this project was to build such a system from scratch, choosing tools, frameworks, and models not based on theoretical novelty, but based on what would actually work well in an educational setting.

Medical students preparing for high-stakes exams like AIIMS, NEET-PG, or USMLE deal with thousands of multiple-choice questions that are often nuanced and clinically detailed. These MCQs demand reasoning across pathology, physiology, diagnostics, and treatment—all in one go. That format presents a challenge not just for learners, but also for system designers trying to support them. I approached this not as a text classification problem in isolation, but as an opportunity to create a practical, modular learning environment that could handle these question types

intelligently, while also being deployable on modest hardware.

Large models like GPT-4 or Med-PaLM have made headlines for their accuracy on medical tasks, but they remain difficult to deploy due to size, cost, and interpretability issues. Instead, I selected PubMedBERT, a smaller but highly domain-aligned model trained entirely on biomedical literature. It provided a strong foundation for classification tasks while remaining light enough to run in a web-based environment. This choice was driven by development needs: low latency, explainable outputs, and full control over the model pipeline.

This thesis introduces MedPredict, a full-stack platform built from the ground up to help medical students learn more effectively through active engagement, immediate feedback, and structured reasoning tools. Every part of the system—from the fine-tuned MCQ classifier and the flashcard engine to the case simulator and frontend interface—was designed and implemented with user experience and deployment feasibility in mind. This is not just a demonstration of what models can do, but an example of how they can be integrated into real tools that real people use.

1.1.2 Motivation

High-stakes medical entrance and licensing examinations such as AIIMS, NEET-PG, and USMLE require candidates to recall thousands of discrete facts and to apply them under time pressure. Existing study resources—PDF question banks, static flash-card apps, and isolated clinical-case books—offer limited feedback and no cross-tool coherence. Students often juggle multiple platforms, wasting time synchronising progress and repeating already-mastered content.

Advances in natural-language processing now allow domain-specific language models (e.g., PubMedBERT) to classify multiple-choice medical questions with near-real-time latency on commodity hardware. Meanwhile, modern JavaScript frameworks (Vue.js) and micro-service back-ends (Node.js + Python) make it straightforward to wrap such models in a responsive, browser-based interface. This convergence creates an opportunity to deliver an interactive, confidence-aware study platform that unifies question answering, flash-card drilling, symptom lookup, and

clinical-case simulation.

The present project, MedPredict, is motivated by the need to close the gap between research-grade biomedical language models and the practical learning tools students actually use. By focusing on usability, interpretability, and low deployment cost, MedPredict aims to improve study efficiency and reduce anxiety for exam candidates, demonstrating how software engineering and NLP can combine to support learning in a high-stakes domain.

1.2 Related Work and Current State of the Art

The use of natural language processing in biomedical and clinical applications has advanced rapidly, particularly with the rise of transformer-based models such as BERT and its domain-specific variants. These models have not only transformed general-purpose language tasks but have also enabled developers to build practical tools in healthcare, education, and diagnostics. For system builders, this has opened new opportunities to integrate AI into applications that assist with information extraction, document tagging, named entity recognition (NER), and most notably, question answering in structured formats.

Within this landscape, medical multiple-choice question answering stands out as a challenging yet valuable use case for educational platforms. It combines the technical requirements of machine learning with the practical needs of students preparing for exams. MCQA tasks in medicine demand that systems understand clinical language, make decisions across subject boundaries, and provide clear and reliable feedback. From a development perspective, this means choosing models, datasets, and interface designs that reflect the complexity of real medical education—not just benchmark performance.

In this section, I highlight the progression of biomedical language models and examine the tools that developers and researchers alike have used to tackle medical MCQA. Special attention is given to the MedMCQA dataset, which this platform is built upon. Unlike general QA datasets, MedMCQA offers realistic exam-style questions drawn from actual medical exams, making it ideal for educational

applications. This section also outlines the implementation strategies such as full fine-tuning versus prompt-based querying that have influenced the system architecture of MedPredict, and explains how this work fits into the broader push to bring NLP-driven tools into real-world learning environments.

1.2.1 Existing Systems

Several platforms and tools currently exist in the space of medical education, ranging from traditional learning apps to AI-powered systems. While these systems offer partial support for exam preparation and content revision, very few provide a fully integrated, interactive environment that combines medical reasoning, MCQ classification, flashcard generation, and clinical simulation in one cohesive platform.

One of the most widely used tools today is Anki, a flashcard-based learning system that is popular among medical students for spaced repetition. While highly customizable, Anki lacks intelligent content generation, domain-aware feedback, or any form of automated reasoning. Users must manually create cards or rely on shared decks, and the system does not adapt to a user's level of understanding or offer contextual guidance.

In the AI domain, ChatGPT and related large language models have become increasingly used informally by students for question answering and concept explanation. However, these tools are general-purpose, not fine-tuned for biomedical vocabulary or exam-specific formats. Their answers can be helpful but are not always consistent or aligned with clinical best practices. Moreover, they do not support MCQ workflows out of the box—such as scoring, answer confidence, or structured feedback—and require manual prompt engineering to simulate a learning environment.

Med-PaLM, developed by Google Research, represents a step toward domain-specific generative models for medical Q&A. It has shown strong performance on multiple medical benchmarks, even reaching or surpassing human performance in some areas. However, its size, infrastructure requirements, and limited public access make it impractical for individual developers, students, or smaller

educational institutions. It is also not available as a deployable product or API, limiting its use as a core component in custom-built educational platforms.

Other tools such as AMBOSS, UptoDate, and Lecturio offer structured medical content, question banks, and analytics for student progress. These platforms, while content-rich, are typically closed systems with little customization, no real-time AI reasoning, and limited transparency in how answers are selected or explained.

In contrast, MedPredict is developed with modularity, interpretability, and interactivity at its core. Unlike closed platforms or generic LLMs, it integrates a domain-specific, fine-tuned MCQ classifier (based on PubMedBERT), a flashcard generator, a rule-based symptom checker, and a clinical decision-making simulator—all accessible through a unified web interface. It is designed not only to answer questions, but to support how students think, revise, and reflect. The backend is fully open and adaptable, enabling future enhancements such as knowledge graph integration, adaptive quizzes, or multilingual support.

Table 1.1 - Comparison of Existing Systems

System	**AI-Powered**	**Open Source / Deployable**	**MCQ Support**	**Clinical Simulation**	**Confidence / Explainability**	**Flashcard Generator**
Anki	✗	✓	✗	✗	✗	✓
ChatGPT	✓	✗	~ manual prompt	✗	~not transparent	✗
Med-PaLM	✓	✗	✓	✗	~ not user-visible	✗
AMBOSS	~ limited	✗	✓	~ (some	✗	✗

	AI			guides)		
Lecturio	~ limited AI	✗	✓	~case-based videos	✗	~ limited decks
MedPredict	✓	✓	✓	✓	✓	✓

✓ = Fully supported ✗ = Not supported ~ = Partially supported or limited

As shown in the comparison above, most existing systems either specialize in one specific area such as flashcard generation or static question banks or rely on general-purpose language models that are not optimized for medical contexts. Tools like Anki offer flexibility but lack AI integration, while platforms like Med-PaLM demonstrate strong medical reasoning but remain inaccessible for deployment or customization. Similarly, systems such as AMBOSS and Lecturio provide curated content but operate as closed ecosystems with limited interactivity and transparency. In contrast, MedPredict offers a balanced solution by integrating domain-specific AI, real-time feedback, and modular design in a fully deployable platform tailored specifically for medical education. This makes it not only technically robust, but also pedagogically relevant and practically usable by students preparing for high-stakes exams.

1.3 Problem Statement

Medical students preparing for high-stakes examinations such as AIIMS, NEET-PG, and USMLE face the overwhelming challenge of mastering vast, multidisciplinary content under intense time constraints. Despite the availability of question banks, flashcards, and online resources, existing tools often operate in isolation and lack intelligent support for clinical reasoning, adaptive feedback, or contextual learning. Current AI-driven solutions either require significant infrastructure (e.g., large-scale models like GPT-4 or Med-PaLM) or fail to provide domain-specific performance, transparency, and deployability. There remains a significant gap in the availability of integrated, modular, and AI-enhanced learning

platforms that are both practically usable and pedagogically aligned with the real-world needs of medical students.

This thesis addresses this gap by developing **MedPredict**, a full-stack web platform that combines domain-specific NLP (via PubMedBERT) with interactive tools such as an MCQ classifier, flashcard generator, clinical case simulator, and a confidence-based feedback engine. The goal is to provide a deployable, interpretable, and student-centered system that supports efficient learning, active recall, and clinical thinking through a unified AI-powered environment.

1.4 Main Work of this Project

The primary objective of this project was to design, develop, and deploy MedPredict a full-stack, AI-powered medical education platform tailored to assist students in mastering multiple-choice questions for competitive medical exams. The main work carried out can be summarized across four key components:

1. **Model Selection and Fine-Tuning**

A domain-specific language model, PubMedBERT, was selected for its alignment with biomedical vocabulary and clinical semantics. It was fine-tuned using the MedMCQA dataset, which contains over 194,000 MCQs from various clinical and preclinical disciplines. The model was optimized to classify four-choice MCQs using supervised learning, with additional outputs for confidence scoring and difficulty estimation. Data preprocessing included tokenization, format normalization, and cleaning of ambiguous entries.

2. **Feature Development and Backend Integration**

The backend was developed using Node.js and Express.js, with Python subprocesses responsible for model inference and logic-heavy components. Core features implemented include:

- An LLM-based MCQ classifier with real-time predictions and probability distribution visualizations.
- A flashcard generator that transforms text content into study-ready decks for active recall.

○ A rule-based symptom checker that simulates preliminary diagnostic reasoning using fuzzy logic.

○ A clinical case simulator structured with JSON-based branching logic, allowing users to navigate realistic, decision-driven case scenarios.

3. **Frontend Design and User Interaction**

The frontend was developed using Vue.js, offering a responsive, modular, and user-friendly interface. Users can toggle between different learning modes (quiz, flashcard, symptom checker, case simulation) and receive instant feedback on their inputs. Visualization components were added for model confidence, question difficulty, and performance tracking.

4. **System Architecture and Modularity**

The system was built with modularity and extensibility in mind. Each feature operates as an independent module communicating via a common API layer. This design enables future upgrades—such as adaptive learning logic, multilingual support, or integration with external medical databases—without major restructuring. The platform was also tested for deployability in local and cloud-based environments.

In summary, this project delivers a complete, functioning educational tool that demonstrates how transformer-based NLP models can be embedded into real-world medical learning platforms. The work bridges the gap between cutting-edge AI models and practical student needs through careful system design, targeted model tuning, and full-stack software development.

1.5 Structure of the Thesis

This thesis is organized into six chapters:

- **Chapter 1: Introduction**

Introduces the context motivation and objectives of the project along with a brief review of related work and the contributions made.

- **Chapter 2: Background**

Surveys prior work in biomedical NLP MCQA modeling, and related LLM strategies, including PubMedBERT, MedMCQA, and domain-specific transfer learning.

- **Chapter 3: System Design**

Defines the functional and non-functional requirements of MedPredict, presents the high-level architecture, and details each core module MCQ classifier, flashcard generator, symptom checker, and clinical-case simulator.

- **Chapter 4: System Implementation**

Details the full implementation pipeline from fine-tuning PubMedBERT to building the backend and frontend infrastructure using Node.js, Python, and Vue.js.

- **Chapter 5: System Test**

Presents testing methodology, performance benchmarks, and qualitative feedback, with analysis of accuracy, confidence scores, and difficulty-based evaluation.

- **Chapter 6: Work Summary and Reflection**

Concludes the thesis with a summary of achievements, a reflection on the challenges encountered, and proposed directions for future improvements and research.

Chapter 2 Background

2.1 Introduction to the Background Knowledge of the MedPredict System

MedPredict is crafted as a comprehensive, end-to-end educational platform that combines state-of-the-art natural language processing with tried-and-true algorithmic methods to deliver a seamless learning experience for medical students. By unifying transformer-based MCQ inference with interactive study tools, the system addresses both knowledge recall and clinical reasoning skills in a single environment. Its architecture is designed to be modular and extensible, allowing each service to evolve independently while sharing a common persistence and API layer. Real-time feedback, confidence visualization, and guided clinical scenarios keep students engaged and help them identify areas for improvement. Moreover, by logging user interactions—such as question responses and flashcard performance—MedPredict can adapt to individual learning curves and provide data-driven insights to both learners and educators. At a high level, the platform comprises five core services:

1. **MCQ Inference Service**
    - Algorithmic Basis: A fine‑tuned PubMedBERT transformer that accepts a concatenated "question + options" string and outputs per‑option logits and confidence scores.
    - Engineering Role: Exposed via a Python CLI (api_mcq_inferrer.py) and wrapped in a Node.js endpoint for real‑time browser queries.
2. **Symptom‑to‑Disease Classifier**
    - Algorithmic Basis: A rule‑and‑fuzzy‑matching pipeline using a synonym dictionary and Levenshtein‑ratio thresholding (fuzzywuzzy) over a CSV of diseases and associated symptoms.
    - Engineering Role: Available in both JavaScript (fast in‑browser) and Python (for consistency), with identical logic for normalizing inputs and scoring disease matches.

3. **Flashcard Maker & Trainer**
    - Algorithmic Basis: Simple parsing of “term;definition” pairs into cards, timed practice rounds, and best‑of‑round repetition tracking.
    - Engineering Role: A Vue component that submits raw text, receives a JSON array of flashcards, and drives a stateful practice session with timing and review logic.
4. **Clinical Case Simulator with GPT Feedback**
    - Algorithmic Basis: Sequential presentation of pre‑authored case steps (JSON), user choice or free‑text input, and an OpenRouter‑backed LLM for end‑of‑simulation feedback.
    - Engineering Role: Managed by two Vue components (CaseSelector.vue & SimulationRunner.vue), coordinating fetch calls to /api/cases/:filename and /api/feedback, then rendering GPT responses inline.

Behind these services sits a **persistence layer** (SQLite for prototyping, PostgreSQL for production), defined via Sequelize models (Session, Question, Response) to track user sessions, questions served, and responses recorded. A **Dockerized Deployment** unifies frontend and backend, ensuring consistent environments and straightforward scalability.

In the chapters that follow, we’ll show how each of these components is designed (ER diagrams, component diagrams), implemented (detailed module specs and code snippets), and assembled into a coherent, deployable system that prioritizes reproducibility, maintainability, and user‑centered interaction.

2.2 Technical Knowledge and Technologies Involved

MedPredict’s engineering stack spans Python for ML, Node.js for orchestration, a relational database for persistence, and Vue 3 for the user interface. Below we detail each technology and its concrete role in the system.

Python 3.10, PyTorch & HuggingFace Transformers

All core ML functionality lives in Python 3.10. We fine-tune PubMedBERT using

PyTorch's Trainer API in training_script.py, specifying hyperparameters such as batch size, gradient accumulation, and mixed precision. For inference, api_mcq_inferrer.py loads the saved weights and AutoTokenizer/AutoModelForSequenceClassificationto process a concatenated "question + options" string into logits. The script then applies torch.softmax to compute per‑option confidence scores, returning a JSON payload via print(json.dumps(...)).

Node.js v18 & Express

The REST API layer is implemented in Node.js (v18) with Express 4.x. We configure middleware—cors(), express.json(), and express.urlencoded()—to handle cross‑origin requests and JSON payloads. Each service (MCQ inference, symptom classification, flashcards, clinical case simulation) is mounted on app.use('/api', routes) in app.js and bound to HTTP routes in routes/index.js.

child_process.spawn & python-shell

To reuse existing Python logic without rewriting it in JavaScript, our Express controllers invoke api_mcq_inferrer.py, symptom_api.py, and the flashcard parser via Node's child_process.spawn or the python-shell module. Output is streamed from Python's stdout, parsed as JSON, and then persisted or forwarded to the client—ensuring a clean separation of concerns between Python ML code and Node.js business logic.

Sequelize ORM with SQLite / PostgreSQL

Persistence is managed via Sequelize: models (Session, Question, Response) are defined in models/index.js with DataTypes.UUID, INTEGER, JSON, and FLOAT fields. In development, we sync these models to a local SQLite file (./data/medpredict.sqlite), while in production the same definitions connect to PostgreSQL via environment‑configured credentials. Foreign‑key relations (Session.hasMany(Response), Response.belongsTo(Question)) enforce referential integrity.

Vue 3, Vue Router & Axios

The front end is a Vue 3 SPA structured into single-file components. We use Options API for simpler pages (e.g. AboutMe.vue) and <script setup> for more complex views

(e.g. SimulationRunner.vue). Vue Router's history mode maps paths (/, /symptoms, /classifier, /flashcards, /simulate) to view components. Axios handles all HTTP interactions: POSTing MCQ payloads to /api/mcq, submitting symptom lists to /api/symptoms-python, and fetching simulation cases from /api/cases/:filename.

pandas & fuzzywuzzy

In the symptom‑to‑disease service (symptom_api.py), we load "DiseaseAndSymptoms.csv" and "Disease precaution.csv" into pandas DataFrames. A synonym dictionary standardizes user inputs before fuzzy matching each against known symptoms using fuzz.ratio. We then compute confidence scores as (matches/total_symptoms)*100 and return the top three diseases along with recommended precautions—all in a single Python script that can be run from either Node.js or as a standalone CLI.

2.3 Biomedical NLP and the Rise of Domain-Specific Transformers

The development of BERT (Bidirectional Encoder Representations from Transformers) by Devlin et al. in 2018 marked a pivotal moment in natural language processing (NLP), introducing a pre-training and fine-tuning paradigm that quickly became the standard. However, its general-purpose nature limited its effectiveness in domain-specific corpora such as biomedical literature. This led to the creation of BioBERT (Lee et al., 2020)**[9]**, a domain-adapted variant of BERT pre-trained on PubMed abstracts and PMC full-text articles. BioBERT achieved notable improvements in biomedical tasks such as named entity recognition (NER), relation extraction, and question answering (QA), thereby highlighting the importance of domain-specific pretraining.

Following this, other specialized models such as SciBERT (trained on multidisciplinary scientific literature) and PubMedBERT Su, P., Peng & Vijay-Shanker (2021)**[19]** were introduced to further enhance transformer performance in focused domains. PubMedBERT, pre-trained exclusively on PubMed abstracts using a biomedical vocabulary from scratch, outperformed previous models in biomedical QA and classification tasks. Its superior syntactic and contextual comprehension made it an ideal candidate for fine-tuning on clinical MCQA datasets such as MedMCQA.

2.4 Summary of This Section

This section has outlined the diverse technical stack behind MedPredict—from deep‑learning frameworks and server‑side orchestration to relational persistence and reactive front-end components. Together, these technologies form a modular, scalable, and maintainable engineering foundation, setting the stage for the detailed system design and implementation chapters that follows.

Chapter 3 System Design

3.1 Functional Requirements and User Goals

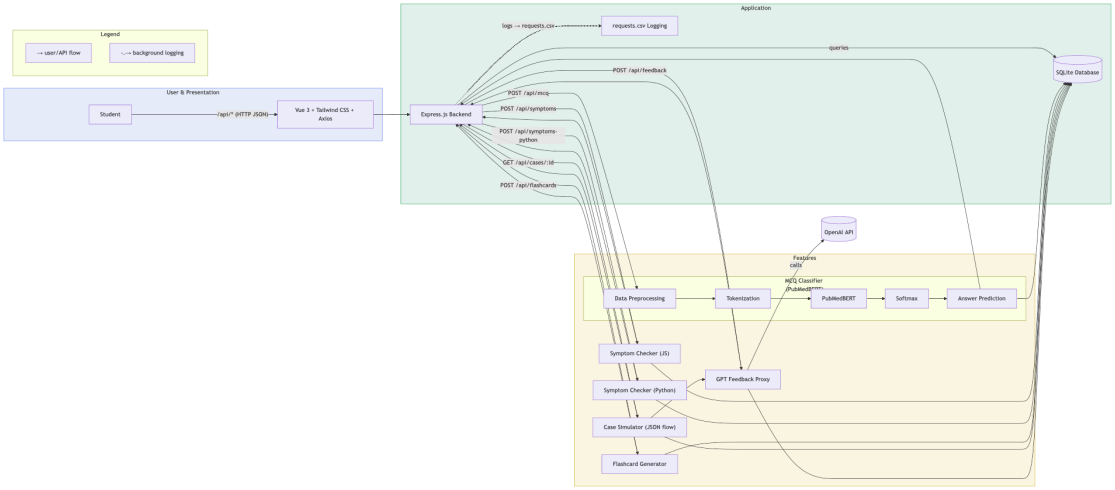


Figure 3.1 MedPredict Application Architecture and End-to-End Data Flow

3.1.1 Educational Objectives

MedPredict was designed as a technical solution to assist medical students with self-testing, concept reinforcement, and clinical scenario simulation. Each tool whether the multiple-choice question interface, flashcard generator, symptom checker, or case simulator was implemented to address a specific user need within medical exam preparation. The system emulates real exam structures and offers immediate feedback through a simplified user interface, enabling repeated use and interaction. These features support more effective user engagement without requiring deep AI or technical knowledge from the end user.

3.1.2 Platform Requirements

MedPredict was developed with clear technical requirements to ensure broad accessibility and reliable performance for its target users—medical students and educators working with standard hardware. The system is optimized to run on personal

laptops without requiring dedicated GPUs or cloud services, supporting use in both online and offline settings.

The platform follows a **modular full-stack architecture**, with components organized into three layers:

- **Local Inference Engine**

The integrated language model (PubMedBERT) is configured for local inference using Python, enabling prediction tasks such as MCQ classification and symptom analysis to run with minimal latency on mid-range CPUs.

- **Modular API Layer**

The backend, built with Node.js and Express.js, exposes RESTful endpoints for major system features. These APIs are decoupled from internal model logic to simplify maintenance and allow for independent upgrades or service extension.

- **Responsive Frontend Interface**

Built using Vue.js and CSS, the frontend is optimized for usability across browsers and screen sizes. The interface supports quiz navigation, flashcard generation, case walkthroughs, and input for symptom checking.

- **Stateless Data Handling**

The system avoids permanent data storage, reducing setup complexity and supporting lightweight use. User session data is stored in memory and can be exported to CSV or JSON format on demand.

In summary, the platform requirements for MedPredict prioritize simplicity, efficiency, and educational value. The goal is to deliver a powerful learning companion that performs well in constrained environments while remaining extensible for future academic or clinical use cases.

3.1.3 User Profiles

MedPredict was developed to support multiple types of users, each with different goals and interaction needs:

- **Medical Students**

The primary users are undergraduate and postgraduate medical students preparing for

standardized exams such as AIIMS, NEET-PG, and USMLE. These users primarily interact with the MCQ quiz system, flashcard generator, and difficulty-based categorization features to guide their practice and content review. The platform requires no setup or accounts, making it easy for students to begin using it immediately on their personal devices.

- **Independent Learners**

Individuals studying outside formal institutions can use MedPredict's symptom checker and content generation tools to create personalized learning sessions. The platform supports lightweight use without needing centralized infrastructure, making it accessible to revision-focused learners.

- **Educators and Facilitators**

Instructors and teaching assistants can leverage the clinical case simulator and MCQ modules during class sessions or tutorials. These tools can assist in demonstrating diagnostic logic, prompting group discussion, or assessing student engagement in a live setting.

While usage patterns vary across profiles, the system's **modular architecture**, **low setup requirements**, and **transparent output presentation** allow all users to interact with the platform according to their specific needs.

3.1.4 Performance Requirements

To maintain a consistent and responsive user experience, MedPredict was designed with several core performance requirements:

- **Low Latency**

All major features—including MCQ classification, simulation progression, and symptom analysis—must return results in **under two seconds** on non-GPU consumer hardware. This ensures real-time responsiveness, which is essential for tool usability and user retention.

- **Efficient Inference and Output Formatting**

The integrated model is configured for local CPU-based inference and optimized for low overhead. Predictions include structured output fields, such as:

  - Per-option confidence scores

- Difficulty level tags (Easy / Medium / Hard)

- These outputs are used to enhance the user interface and support optional feedback features, not for model evaluation or performance benchmarking.

- **Cross-Platform Compatibility**

The system was tested across multiple browsers and operating systems to ensure consistent behavior. Components were developed with low-resource environments in mind.

By adhering to these performance standards, MedPredict provides a technically stable, responsive, and extensible tool that aligns with common hardware and user interaction expectations.

3.1.5 Requirements Summary

Table 3.1 : Functional and Non-Functional Requirements of MedPredict

Req. ID	Type	Description
FR1	Functional	Users can submit a four-option MCQ and receive per-option logits, confidence, and difficulty tag.
FR2	Functional	Users can convert “term;definition” lines into flashcards, run timed study rounds, and repeat unmastered cards.
FR3	Functional	Users can input symptoms (text list) and receive the top three diagnoses with confidence scores.
FR4	Functional	Users can walk through JSON-driven clinical cases step by step and receive GPT-based feedback.
NFR1	Non-Functional	All API calls (MCQ, symptoms, flashcards, cases) must respond within 2 s on a standard CPU-only laptop.
NFR2	Non-Functional	System must run locally with zero external dependencies (no cloud GPU or managed databases required).

NFR3	Non-Functional	Front end must support modern desktop and mobile browsers (Chrome, Firefox, Safari).
NFR4	Non-Functional	Codebase must be modular and testable: each service (MCQ, symptom, flashcards, simulation) has its own controller and tests.

3.2 Module Overview

MedPredict consists of multiple interdependent modules, each responsible for a distinct aspect of the system's functionality. These modules are designed to work together through a shared API layer, allowing modular development, independent updates, and streamlined user experience.

3.2.1 Admin Module

The Admin Module is intended for academic staff or platform maintainers responsible for managing the MedPredict environment. It enables administrative control over data, user access, and platform configuration.

- Access to the Admin Module is restricted to authorized personnel through credential-based authentication.
- The admin dashboard provides a centralized view of system usage, including flashcard activity, MCQ submission history, and active user sessions.
- The Manage MCQ Dataset feature allows administrators to upload, remove, or categorize MCQs used within the quiz engine.
- The Manage Flashcards function enables viewing, approval, or deletion of auto-generated flashcards submitted by users.
- Through the User Access Control panel, administrators can approve new accounts, assign roles (student, educator, etc.), and revoke access as needed.
- Logs and basic analytics provide transparency into system interactions and usage trends.

This module ensures that the platform remains consistent, secure, and aligned with

institutional or course-specific requirements.

3.2.2 User Module (Student)

The User Module is designed for medical students who are the primary users of the MedPredict system. It supports interactive learning through AI-powered tools integrated into a unified web interface.

- Students can log in with their credentials and access tools such as the MCQ quiz engine, flashcard generator, case simulator, and symptom checker.
- The MCQ Engine allows users to submit questions and receive predictions along with confidence scores and difficulty labels.
- The Flashcard Tool enables students to convert text-based content into customizable flashcards for revision.
- The Symptom Checker allows users to input clinical symptoms and receive potential diagnoses based on fuzzy rule-matching logic.
- The Case Simulator guides users through step-by-step clinical scenarios using a JSON-driven branching structure.
- Students can view previous interactions, export flashcard decks, and track their confidence-based performance over time.

This module is optimized for usability, performance, and fast feedback to facilitate high-volume practice and low-friction revision.

3.3 System Architecture Design

3.3.1 Modular Design

The architecture of MedPredict follows a modular design that separates concerns across distinct layers. The system is divided into four primary modules. The frontend layer manages all user interactions and interface components. The backend layer acts as a controller that routes requests and integrates logic. The inference engine handles model predictions and data processing. Finally, the data layer supports persistence and caching.

This separation ensures that each component can be developed maintained and scaled independently without affecting the others. It also allows greater flexibility in upgrading

individual features such as replacing the model or updating the frontend design.

3.3.2 Use Case Diagram

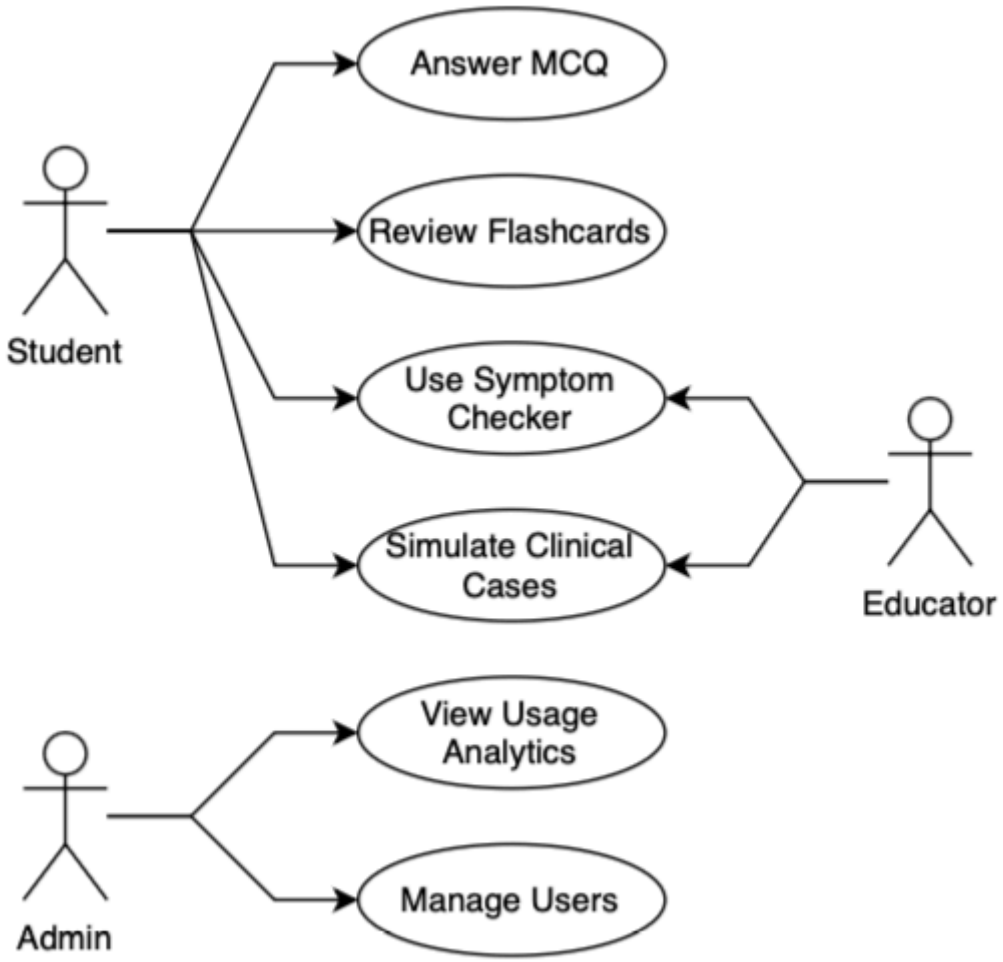


Figure 3.2: Use Case Diagram of the MedPredict Platform

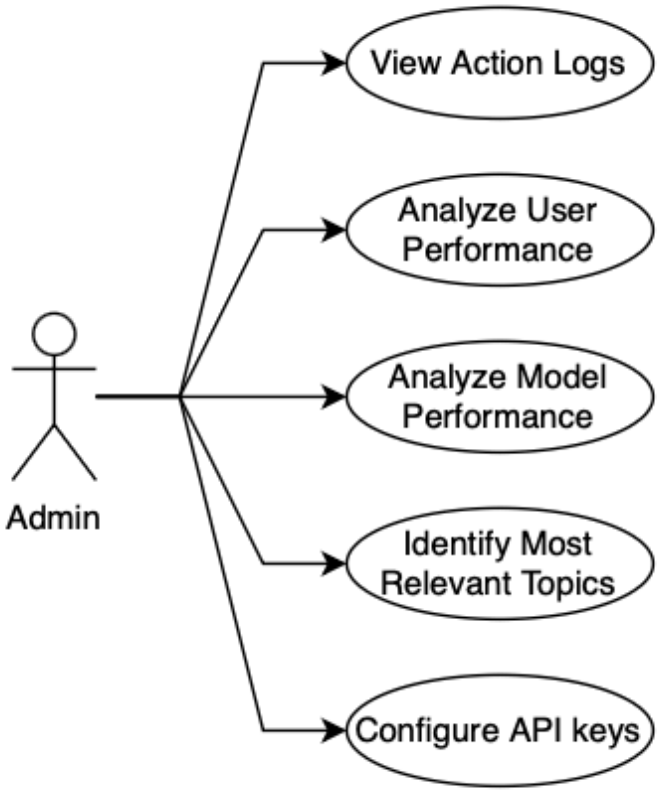


Figure 3.3: Use Case Diagram for Admin & Maintenance

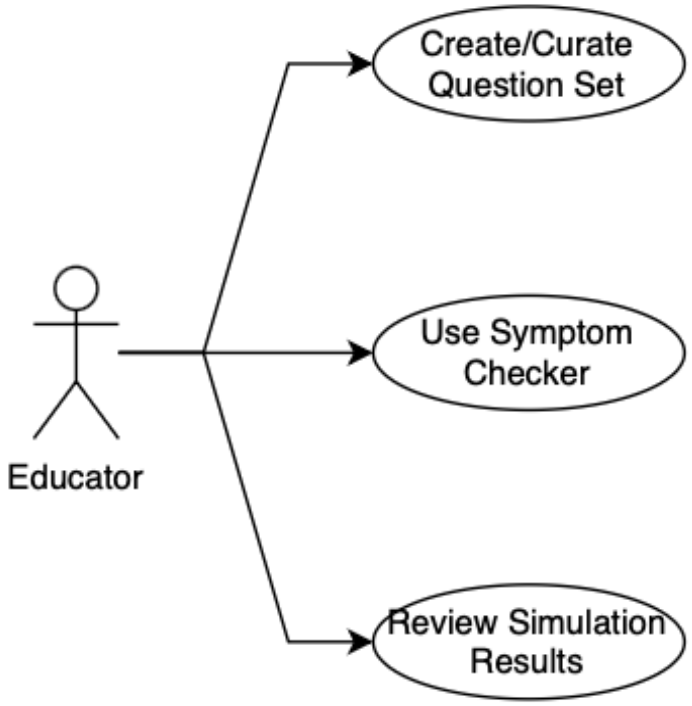


Figure 3.4: Educator-Centric Diagram

3.3.3 Layer Responsibilities

This UML use case diagram illustrates the primary actors Medical Student and Educator and their interactions with MedPredict's core functions. The Medical Student can Answer MCQs, Review Flashcards, Use the Symptom Checker, and Simulate Clinical Cases, while the Educator may also guide and review student progress through these same services. Associations between actors and use cases show how each user role engages with the system's educational tools.

- **Frontend**

The frontend is built using VueJS. It presents users with a clean and interactive interface where they can access different features such as quizzes simulations flashcards and symptom checking. Each section is organized into Vue components that handle form input output display and routing. The design emphasizes usability and responsiveness to support access across different devices.

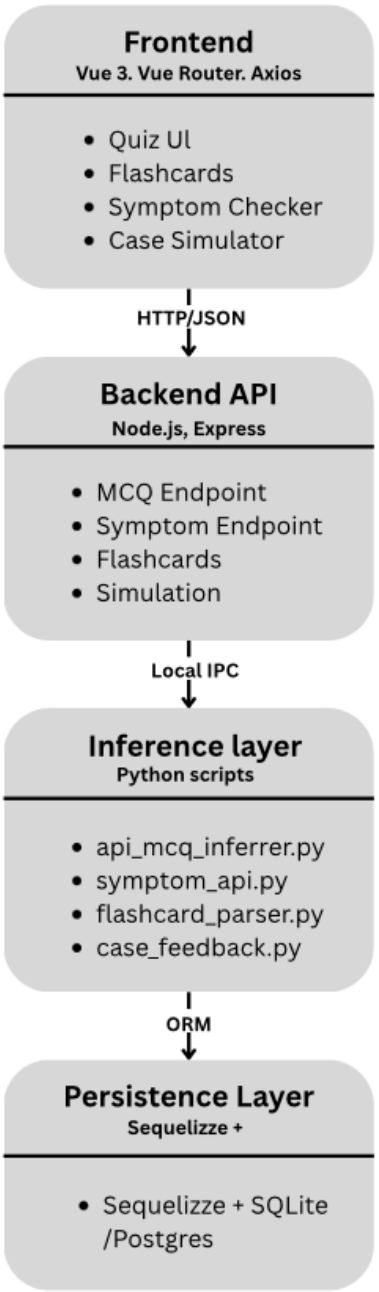
- **Backend**

The backend is built using ExpressJS. It serves as the central API layer that receives

requests from the frontend routes them to the correct processing modules and returns results. It also manages HTTP communication handles validation and ensures secure logic flow. For heavy tasks like model predictions it delegates processing to Python-based subprocesses.

- **Inference Layer**

The inference layer is written in Python. It executes machine learning logic such as MCQ classification using PubMedBERT symptom classification via rule-based or fuzzy logic and GPT-based feedback generation for clinical simulations. Each Python script is triggered by the backend and returns structured JSON responses for display in the frontend.



- **Data Layer**

The Data Layer is a fully local subsystem that handles storage, configuration, caching, and logging without external services. It uses an embedded SQLite database for core entities (users, flashcards, symptom logs) with ACID transactions and cascading deletes. Configuration and seed data load from JSON/CSV files

MedPredict consists of the following key modules:

- **MCQA Classifier**

A PyTorch-based PubMedBERT model fine-tuned to classify medical questions into one of four answer options (A–D). This model serves as the core decision engine for MCQ inference.

- Interactive Inference Engine

A terminal-based tool that accepts user-input questions and choices, predicts the answer, estimates difficulty,

Figure 3.5: Four-Layer Software Architecture of MedPredict

- and visualizes confidence probabilities. This module also supports quiz mode and logs user interactions for performance analysis.

- **Symptom Checker**

A rule-based Python engine using fuzzy matching on medical CSV datasets to suggest likely conditions based on user-input symptoms. It returns the top three differential diagnoses along with precautionary advice.

fuzzy matching logic :

For two strings A and B:

FuzzyMatch(A,B)=fuzz.ratio(A,B)

(threshold > 85%)

- **Clinical Case Simulator**

A JSON-driven simulator that guides users through multi-step clinical scenarios. The tool supports branching logic, decision tracking, and reflective feedback, with optional integration of GPT-based summary support.

- **Flashcard Module:**

A simple input-output system for entering, reviewing, and tracking

flashcards in "term; definition" format. Includes time tracking and memorization rounds.

- **Frontend:**

Built using Vue.js, the UI includes multiple feature routes: home, quiz, case

simulation, flashcards, and a symptom checker page. Each is designed with usability and

responsiveness in mind.

- **Backend:**

Powered by Express.js, the backend manages API routes for all tools, including

Python model integration via child processes. It connects with static files and handles client-server communication.

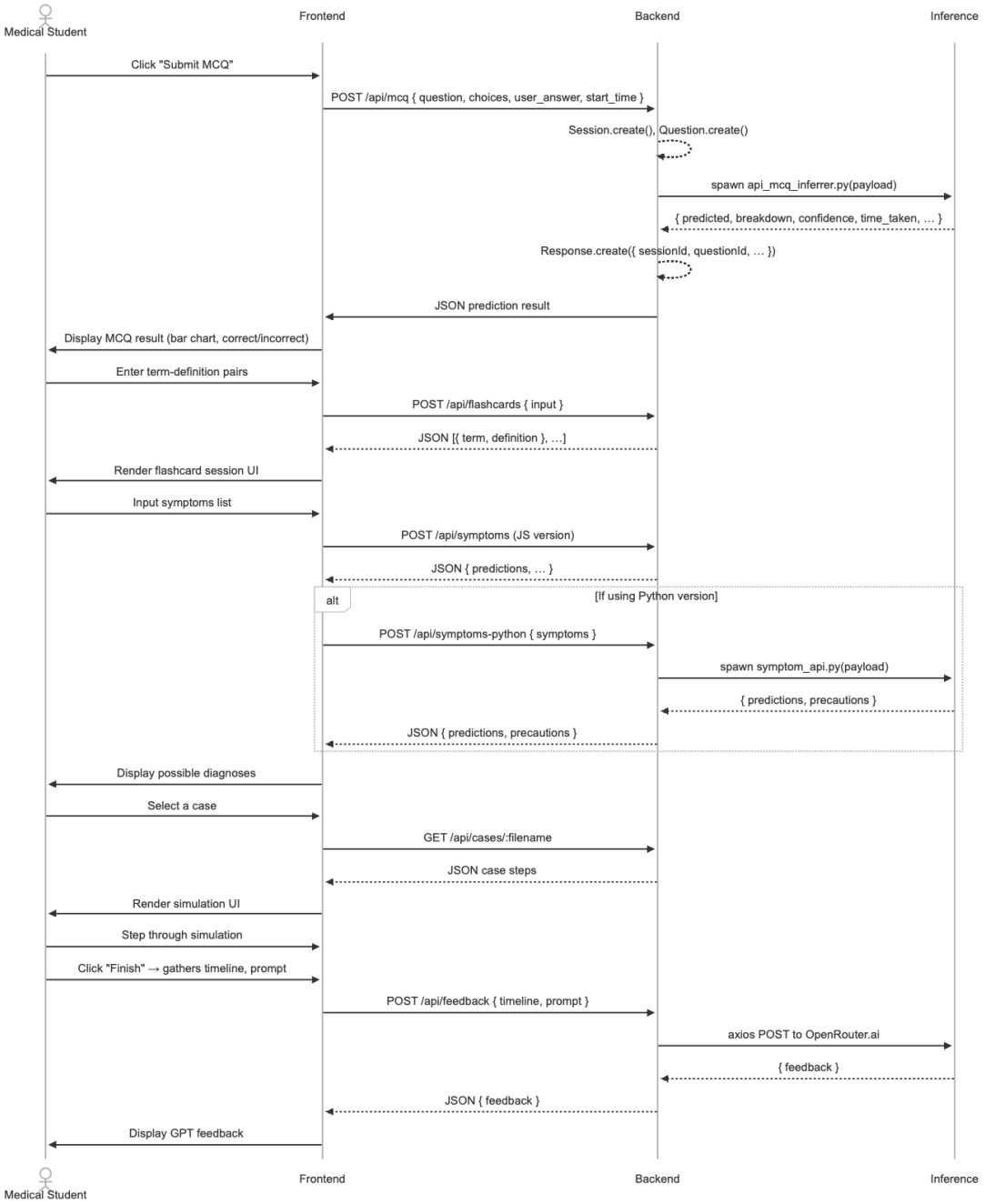


Figure 3.6: Sequence Diagram

Figure 3.4 illustrates the end-to-end message flow among the four principal components of MedPredict Medical Student, Frontend, Backend, and Inference across its core use cases. Each vertical dashed line denotes the “lifeline” of one component, while horizontal arrows indicate the sequence of requests, responses, and internal actions that implement each feature. The diagram is organized into four distinct sections:

MCQ Inference, Flashcard Generation, Symptom Checker, and Simulation & GPT Feedback.

MCQ Inference Flow

1. **Student → Frontend**
   The student clicks the Submit MCQ button in the learning interface.
2. **Frontend → Backend**
   The browser issues a POST /api/mcq containing the question text, choice set, the student's selected answer, and a timestamp.
3. **Backend (internal)**
   The server immediately persists a new session record and associated question record in the database.
4. **Backend → Inference**
   It then spawns the Python inference worker (api_mcq_inferrer.py) with the payload.
5. **Inference → Backend**
   The worker returns a JSON object with the predicted answer, a detailed breakdown of its reasoning, a confidencescore, and time taken.
6. **Backend (internal)**
   The server logs that response by creating a new Response entry linked to the session and question.
7. **Backend → Frontend → Student**
   Finally, the JSON prediction result is delivered back to the browser, which renders a bar chart comparing correct and incorrect options and highlights the student's performance.

Flashcard Generation Flow

1. **Student → Frontend**
   The student enters raw "term–definition" pairs into a simple text area.
2. **Frontend → Backend**
   A POST /api/flashcards call sends that input to the server.

3. **Backend → Frontend → Student**
   The server parses the input into a JSON array of discrete { term, definition } objects, which the UI then uses to render an interactive flashcard study session.

Symptom Checker Flow

1. **Student → Frontend**
   The student submits a list of symptoms.
2. **Frontend → Backend**
   By default, the UI invokes a JavaScript–based endpoint (POST /api/symptoms), which synchronously returns preliminary { predictions, … }.
3. **Alternate Python Path**
   If the Python implementation is selected, the front end instead calls POST /api/symptoms-python. The backend forwards this to symptom_api.py in the Inference layer, then returns the richer { predictions, precautions }result.
4. **Frontend → Student**
   The UI presents the possible diagnoses (and any associated precautions) for the student's review.

Simulation & GPT Feedback Flow

1. **Student → Frontend**
   The student chooses a clinical case from the list.
2. **Frontend → Backend**
   A GET /api/cases/:filename retrieves the JSON-encoded sequence of case steps, which the UI uses to render a step-by-step simulation.
3. **Student → Frontend**
   As the student works through and ultimately clicks Finish, the browser collates the entire interaction timeline along with a feedback prompt.
4. **Frontend → Backend → Inference**
   The server relays this to OpenRouter.ai via an Axios POST. GPT returns structured feedback on the student's clinical decision-making.

5. **Backend → Frontend → Student**
   That feedback is delivered to the UI, closing the loop with personalized, AI-generated commentary.

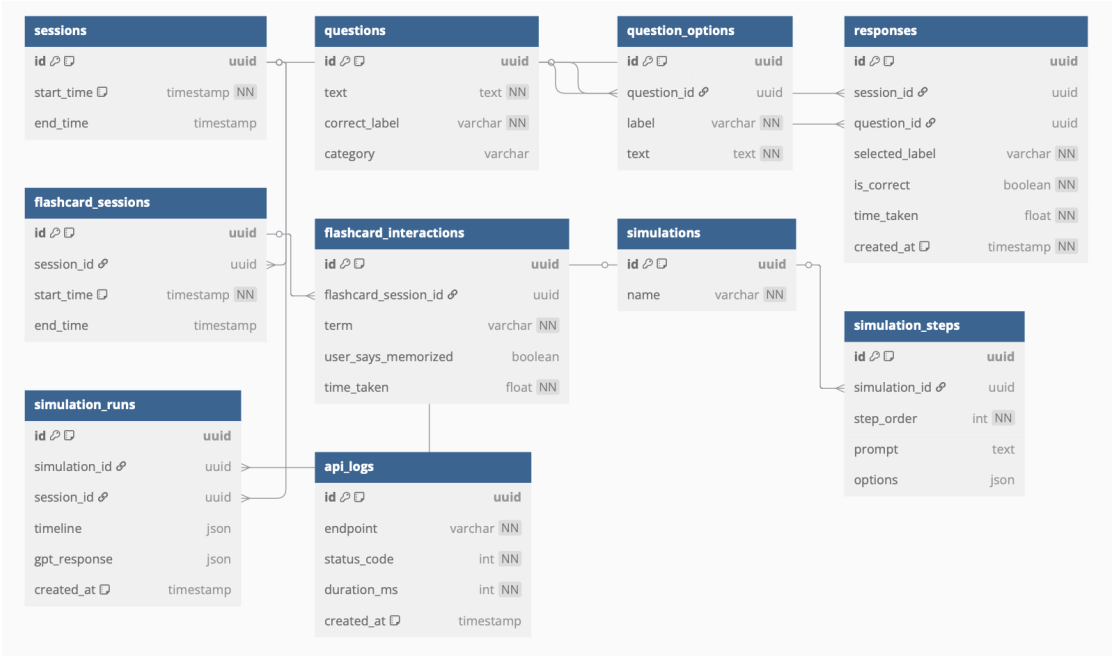


Figure 3.7: Entity-Relationship Diagram of the MedPredict Database

The diagram above models all of the core persistence entities in the MedPredict system. At its heart are:

- **sessions 1— responses*:** each MCQ or simulation run is a Session; a Session can have many Responses (one per question answered).
- **questions 1— responses*:** every Response points back to the Question it answered, preserving correctness and timing.
- **questions 1— question_options*:** each Question has a fixed set of labeled options (A–D).
- **flashcard_sessions 1— flashcard_interactions*:** groups user interactions with flashcards into distinct sessions, recording term, response, and timing.
- **simulations 1— simulation_steps*:** defines reusable case workflows as ordered steps with prompts and option payloads.

- **simulation_runs:** ties a particular run (timeline of choices + GPT feedback) back to both a Simulation and a Session.
- **api_logs:** a flat audit table capturing every API endpoint call, status code, and duration.

tables **quizzes** and **quiz_questions** (a many-to-many join between Quizzes and Questions) support future grouping of questions into named quizzes. although this feature is still in process of implementation into the front end

Rationale for Relationships:

- 1-to-many (hasMany) was chosen wherever a parent entity naturally aggregates multiple child records (one Session → many Responses).
- Join table quiz_questions allows a Question to appear in multiple Quizzes without duplication (and to preserve a custom ordering via the order field).
- Cascade delete on all foreign-key relationships ensures that, for example, removing a Session will automatically remove its associated Responses, FlashcardSessions, and SimulationRuns—preventing orphaned records.
- Breaking out SimulationSteps from SimulationRuns allows the static case definition (steps and prompts) to remain separate from each user's dynamic run data (timeline + GPT response)

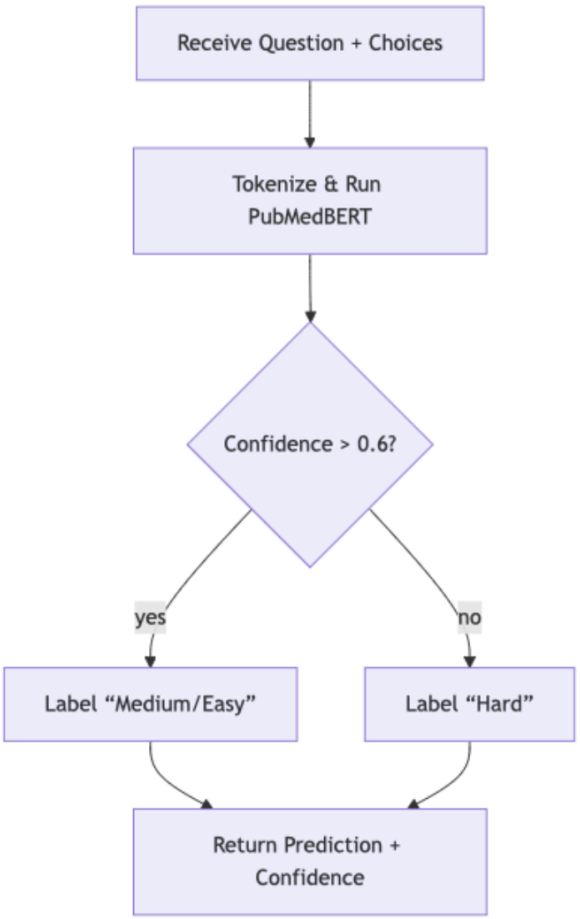


Figure 3.8 Activity Diagram: MCQ Inference Workflow

3.4 Summary of This Chapter

This chapter detailed the architectural and design foundations of MedPredict, presenting its modular structure across frontend, backend, inference, and data layers. It began by outlining the educational goals, user profiles, and technical requirements that guided system development. The chapter then broke down the responsibilities of each system layer, showing how the Vue.js frontend, Express.js backend, and Python-based inference logic work together to deliver an interactive and efficient learning platform. The MCQA classifier, flashcard tool, symptom checker, and clinical simulator were highlighted as core components designed to enhance self-assessment, reasoning, and

feedback. Emphasis was placed on modularity, transparency, and responsiveness to support future extensibility. Overall, the system was designed to balance high educational value with technical scalability, laying the groundwork for implementation discussed in the next chapter.

Chapter 4 System Implementation

4.1 System Overview & Technology Stack

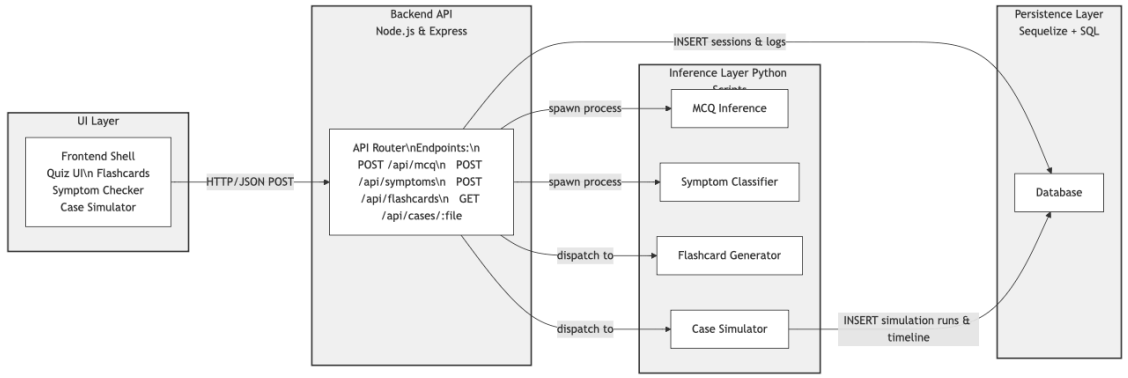


Figure 4.1: Layered Data-Flow Architecture of the MedPredict Platform

4.1.1 Purpose and scope

MedPredict is a fully dynamic web environment combining a Vue 3 frontend with a Node/Express backend and SQLite/Sequelize persistence. It delivers four interactive learning tools flashcards for active recall, a PubMedBERT-powered MCQ classifier, a step-by-step clinical simulation, and a symptom-to-disease matcher with precautions.

4.1.2 Languages & Frameworks

Table 4.1: Overview of the MedPredict Technology Stack

Layer	**Language / Framework**	**Purpose**
Backend	Node.js / Express Python 3.10	HTTP API gateway, routing, middleware Inference workers (MCQ & Symptom scripts)

	SQLite + Sequelize (ORM)	Lightweight on-disk persistence
Frontend	Vue.js 3 + Composition API	Single-page application
Data	Vue Router Axios CSV JSON	Client-side routing HTTP client Symptom & precaution tables Clinical case definitions

4.1.3 Build & Tooling

Node.js / npm

backend/package.json scripts:

- npm run dev → starts Express with live-reload
- npm test → runs Jest/Supertest on controllers

frontend/package.json scripts:

- npm run serve → launches Vite dev server
- npm run build → outputs production assets

Vite (used by Vue 3)

Fast HMR during development

Tree-shaking and code-splitting in production

Prettier & ESLint

Enforce consistent code style across JS, Vue, and Python

4.2 Backend Implementation

4.2.1 API Gateway & Routing

The backend is implemented as a thin Node.js (Express 4) layer whose sole responsibility is to orchestrate requests, delegate to workers, and return JSON results. Key characteristics:

Listing 4.1 Core API Route Definitions

```
// Symptom Classifier (JS)
router.post('/symptoms', classifySymptoms);
// Symptom Classifier (Python)
router.post('/symptoms-python', runPythonSymptomClassifier);
// Flashcard Generator
router.post('/flashcards', generateFlashcards);
// MCQ Classifier
router.post('/mcq', classifyMCQ);
// Quiz Management
router.post('/quizzes', createQuiz);
router.get('/quizzes', listQuizzes);
router.get('/quizzes/:id', getQuiz);
// Clinical Case Loader
router.get('/cases/:filename', (req, res) => {
  const filename = req.params.filename;
  const casePath = path.join(__dirname, '..', '..', 'data', 'cases', filename);
  if (!fs.existsSync(casePath)) {
    return res.status(404).json({ error: 'Case file not found' });
  }
  res.sendFile(casePath);
});
// GPT Feedback Proxy
router.post('/feedback', async (req, res) => {
  const { timeline, prompt } = req.body;
```

```
25. const formattedTimeline = timeline
26.   .map((t, i) => `Step ${i+1}: ${t.choice || t.test || t.input}`)
27.   .join('\n');
28. const fullPrompt = `${prompt}\n\nTimeline:\n${formattedTimeline}`;
29. try {
30.   const response = await axios.post(
31.     'https://api.openrouter.com/v1/chat/completions',
32.     { model: 'gpt-4', messages: [{ role: 'user', content: fullPrompt }] },
33.                                   { headers: { Authorization: `Bearer
    ${process.env.OPENROUTER_API_KEY}` } }
34.   );
35.   res.json(response.data);
36. } catch (error) {
37.   res.status(500).json({ error: 'Failed to fetch feedback' });
38. }
39. });
```

Route Granularity

Each pedagogical feature has its own endpoint under /api:

POST /api/mcq	MCQ inference
POST /api/symptoms	JS‑based symptom matcher
POST /api/symptoms-python	Python‑based symptom matcher
POST /api/flashcards	Flashcard generator
GET /api/cases/:Case name	Clinical case loader
POST /api/feedback	GPT feedback proxy

- **Process Isolation**

  For Python‑powered features (mcq, symptoms-python, mini-quiz), the route handler uses child_process.spawn('python3', [...]), passing the JSON payload on the command line. Standard output is streamed back to Express, so any crash or hang affects only that request handler, not the entire server.

- **Twelve-Factor Configuration**
  All mutable settings model directory path, OPENROUTER_API_KEY, PORT are injected via environment variables. This allows the same Docker image or codebase to run unmodified in development, staging, or production.

Lightweight Observability

A custom middleware logs each request to logs/requests.csv with:
timestamp, method, route, status, latency_ms

- This flat‑file approach preserves a stateless API design while giving enough data to troubleshoot performance or error trends.

4.2.2 Vue 3 Client Application

The frontend is a single‑page application built with Vue 3 (Composition API) and Tailwind CSS. It consumes the REST API via Axios and presents five interactive modules:

1. **MCQ Quiz Panel**
   - Four‑option form bound to ClassifierQuiz.vue
   - submit() posts to /api/mcq, then renders coloured confidence bars, predicted label, difficulty badge, and timing metrics
   - Export buttons allow download of JSON or CSV session data
2. **Symptom-Checker Widget**
   - SymptomChecker.vue accepts comma-separated symptoms
   - Chooses between /api/symptoms (JS) or /api/symptoms-python (Python) based on user selection
   - Displays top-3 diagnoses and precaution list in card layout
3. **Flashcard Generator**
   - FlashcardTrainer.vue parses user “term;definition” input
   - Posts to /api/flashcards, then renders flip‑card UI with timing and memorization toggles
4. **Clinical-Case Player**

- CaseSelector.vue lists available cases/*.json; SimulationRunner.vue steps through each prompt
- Collects a timeline of choices/free-text, then posts to /api/feedback for GPT‑generated feedback

4.2.3 Data Modeling & Persistence

The MedPredict backend uses **SQLite** as a lightweight, file-based datastore, accessed via the **Sequelize** ORM. All data lives in data/medpredict.sqlite, which is created (or synced) automatically on startup.

Model Definitions

- **Session**: tracks each MCQ or flashcard session
- **Question** & **QuestionOption**: store MCQ text and options
- **Response**: records student answer, model prediction, correctness, and time taken
- **FlashcardSession** & **FlashcardInteraction**: log each flashcard round and user memorization decisions
- **Simulation**, **SimulationStep**, **SimulationRun**: define clinical case flows and record student timelines + GPT feedback
- **ApiLog**: lightweight request logging

Each model is defined in models/index.js with a UUID primary key and the appropriate fields (e.g. selected_label, is_correct, time_taken on **Response**). Associations mirror our sequence flows:

Listing 4.2: Sequelize Model Definitions

```
// === Question ↔ QuestionOption ===
Question.hasMany(QuestionOption, {
  foreignKey: 'question_id',
  onDelete: 'CASCADE'
});
QuestionOption.belongsTo(Question, {
  foreignKey: 'question_id'
});

// === Session ↔ Response ===
```

```
Session.hasMany(Response, {
  foreignKey: 'session_id',
  onDelete: 'CASCADE'
});
Response.belongsTo(Session, {
  foreignKey: 'session_id'
});

// === Question ↔ Response ===
Question.hasMany(Response, {
  foreignKey: 'question_id',
  onDelete: 'CASCADE'
});
Response.belongsTo(Question, {
  foreignKey: 'question_id'
});

// === FlashcardSession ↔ FlashcardInteraction ===
FlashcardSession.hasMany(FlashcardInteraction, {
  foreignKey: 'flashcard_session_id',
  onDelete: 'CASCADE'
});
FlashcardInteraction.belongsTo(FlashcardSession, {
  foreignKey: 'flashcard_session_id'
});

// === Simulation ↔ SimulationStep ===
Simulation.hasMany(SimulationStep, {
  foreignKey: 'simulation_id',
  onDelete: 'CASCADE'
});
SimulationStep.belongsTo(Simulation, {
  foreignKey: 'simulation_id'
});

// === Simulation ↔ SimulationRun ===
Simulation.hasMany(SimulationRun, {
  foreignKey: 'simulation_id',
  onDelete: 'CASCADE'
```

```
});
SimulationRun.belongsTo(Simulation, {
  foreignKey: 'simulation_id'
});

// === Session ↔ SimulationRun ===
Session.hasMany(SimulationRun, {
  foreignKey: 'session_id',
  onDelete: 'CASCADE'
});
SimulationRun.belongsTo(Session, {
  foreignKey: 'session_id'
});

// === Session ↔ ApiLog ===
Session.hasMany(ApiLog, {
  foreignKey: 'session_id',
  onDelete: 'CASCADE'
});
ApiLog.belongsTo(Session, {
  foreignKey: 'session_id'
});
```

Excerpt of the sequelize.define(…) calls for Session, Question, QuestionOption, and Response, showing table names, primary keys, and fields.

Listing 4.3: Sequelize Entity Associations

```
// === Session Model ===
const Session = sequelize.define('Session', {
  id: {
    type: DataTypes.UUID,
    defaultValue: DataTypes.UUIDV4,
    primaryKey: true
  },
  start_time: {
    type: DataTypes.DATE,
    allowNull: false
  },
```

```
  end_time: {
    type: DataTypes.DATE,
    allowNull: true
  }
}, {
  tableName: 'sessions',
  timestamps: false
});
```

Excerpt of the Model.hasMany(…) / Model.belongsTo(…) calls illustrating the MCQ, flashcard, and simulation relationships (sessions→responses, questions→options, sessions→flashcard sessions, simulations→steps/runs).

MedPredict persists all of its application data in a single SQLite database file (medpredict.sqlite) located in the data/directory. At runtime, Sequelize automatically creates any missing tables without overwriting existing records by invoking sequelize.sync({ force: false }). A quick inspection of the live schema—via SQLite's PRAGMA table_info(...) commands—confirms that each table's columns exactly match our model definitions. For example, the sessions table uses a UUID primary key and stores precise start_time and optional end_time timestamps; the questions table holds the MCQ text, an optional correct-answer label, and an optional category field; and question_options contains four rows per question, each pairing a choice label (A–D) with its display text. Student interactions are captured in the responses table, which links back to both sessions and questions via foreign keys, and records the user's selected label, a correctness boolean, the time taken to answer, and a creation timestamp. Flashcard study is similarly tracked by two tables: flashcard_sessions groups a run of term-definition drills, and flashcard_interactions logs each individual card's term, the "memorized" flag if set, and the time spent. Clinical-case logic lives in three tables simulations defines the available case names, simulation_steps enumerates each step's prompt and optional JSON‑encoded choices, and simulation_runs captures a JSON timeline of the student's decisions together with the GPT-generated feedback. Finally, every HTTP request is logged to api_logs with endpoint, status code, and duration, providing a lightweight audit

trail without compromising statelessness. Cascade delete rules on all associations ensure that, for instance, deleting a session will automatically remove its associated responses, flashcard sessions, and simulation runs, thus maintaining referential integrity with zero manual cleanup. This file-based design keeps MedPredict easy to deploy, backup, and inspect anyone can open the database in a standard SQLite browser to verify its exact structure and contents.

Listing A.1 SQLite Table Schema Introspection via PRAGMA

```
cd backend/data
sqlite3 medpredict.sqlite <<'SQL'
.headers on
.mode column
PRAGMA table_info('sessions');
PRAGMA table_info('questions');
PRAGMA table_info('question_options');
PRAGMA table_info('responses');
PRAGMA table_info('flashcard_sessions');
PRAGMA table_info('flashcard_interactions');
PRAGMA table_info('simulations');
PRAGMA table_info('simulation_steps');
PRAGMA table_info('simulation_runs');
PRAGMA table_info('api_logs');
SQL
```

Figure 4.2 Series of PRAGMA table_info(...) commands in the SQLite shell.

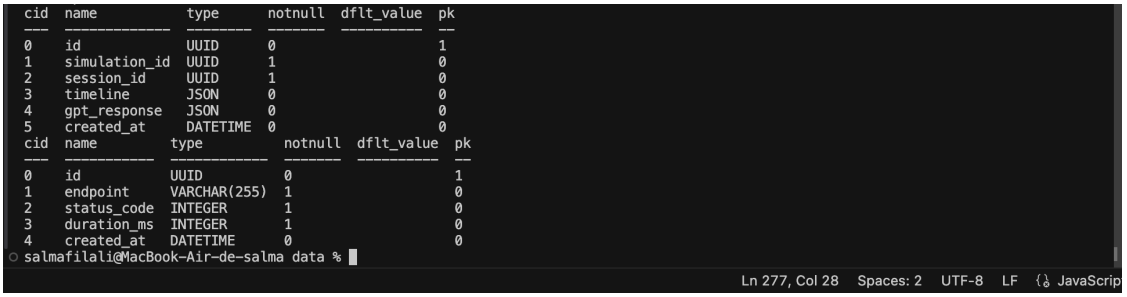


Figure 4.3 Output of PRAGMA table_info(...) in SQLite showing the column names, types, nullability, and primary‑key flags for all core tables

As you can see, each table's columns exactly match our Sequelize definitions UUID primary keys, DATETIME defaults, and foreign-key constraints. There are no extraneous tables or columns, confirming our schema is correctly synchronized.

4.3 Main Modules Implementation

4.3.1 Backend API Services

I built the backend using Node.js and Express.js and structured it around RESTful API

endpoints for each feature. The main route is /api/mcq which handles the PubMedBERT-based question answering task and returns results with confidence difficulty and response time. Other endpoints support the symptom checker mini quiz extraction flashcard generation textbook search and simulation feedback generation.

To keep the system flexible and lightweight I implemented the core logic in Python and called it from the Node.js backend using child process execution. This separation let me update or replace Python logic without modifying the server architecture. It also helped avoid memory and performance issues during inference.

- **/api/mcq** accepts a question and four choices and returns MCQ predictions using the PubMedBERT model. It provides confidence scores, difficulty level, correctness, and response time
- **/api/symptoms** runs a JavaScript-based symptom classifier that uses fuzzy matching on symptom CSV files to suggest possible conditions
- **/api/symptoms-python** connects to a Python script that performs more advanced differential diagnosis using a rules-based approach
- **/api/textbook/search** takes in an uploaded PDF and a keyword and returns relevant page previews based on full-text search (this feature has not been implemented to frontend but its fully implemented on the backend)
- **/api/mini-quiz** generates a short quiz by extracting questions from a selected textbook page (this feature has not been implemented to frontend but its fully implemented on the backend)
- **/api/flashcards** turns term;definition inputs into flashcards that students can review
- **/api/cases/:filename** loads a clinical simulation scenario stored as a JSON file
- **/api/feedback** sends simulation timeline data to OpenRouter and returns GPT-based feedback to help students understand their decisions

For routes that use Python logic like symptom analysis and simulation feedback, I used child process integration to run external scripts. This setup allows smooth communication between the Node.js backend and Python inference modules without blocking the system.

4.3.2 Frontend Development

I used Vue.js and standard CSS to build the frontend. I chose Vue.js for its reactivity and ease of integration with backend APIs. The frontend includes a home page with icons

for each feature like Symptom Checker Flashcards Quiz and Simulations. I made the layout responsive so it works well on both desktops and mobile devices.

One of my goals was to make the interface intuitive and fast. Each feature has a clear layout and results are displayed with visual aids such as confidence bars and labels. For example in the MCQ quiz component the answer is shown along with the confidence and difficulty label. In the flashcard and simulation modules users interact step by step which helps them stay focused. I also used Axios for all HTTP communication with the backend.

To enhance the user experience I ensured results appear instantly after submission and allowed students to interact with the system without needing to reload pages. This fluid interaction design helped maintain engagement and supported repeated use.

4.3.3 Symptom Checker Logic

The rule-based symptom checker is powered by two structured CSV files:

- **DiseaseAndSymptoms.csv** maps each disease to a list of associated symptoms spread across multiple columns
- **Disease_precaution.csv** provides corresponding precautionary steps for each disease

Listing 4.4: Python‑Based Symptom Classifier Implementation

```
const runPythonSymptomClassifier = (req, res) => {
  const symptoms = req.body.symptoms;
  if (!Array.isArray(symptoms)) {
    return res.status(400).json({ error: "Symptom list is required." });
  }

  const pythonProcess = spawn("python3", [
    "./archive/symptom_api.py",
    JSON.stringify(symptoms)
  ]);

  let output = "";
  let errorOutput = "";

```

```
15.  pythonProcess.stdout.on("data", (data) => {
16.    output += data.toString();
17.  });
18.
19.  pythonProcess.stderr.on("data", (data) => {
20.    errorOutput += data.toString();
21.  });
22.
23.  pythonProcess.on("close", (code) => {
24.    if (errorOutput) {
25.      console.error("Python stderr:", errorOutput);
26.      return res
27.        .status(500)
28.        .json({ error: "Python script error", detail: errorOutput });
29.    }
30.    try {
31.      const parsed = JSON.parse(output.trim());
32.      return res.json(parsed);
33.    } catch {
34.      console.error("Failed to parse:", output);
35.      return res.status(500).json({ error: "Invalid JSON from Python" });
36.    }
37.  });
38. };
```

Listing 4.5: Symptom Checker API Request Handler (Vue.js Frontend)

```
1.  async checkSymptoms() {
2.    console.log("Button clicked");
3.    if (!this.symptoms.trim()) return;
4.
5.    try {
6.      const response = await fetch("/api/symptoms-python", {
7.        method: "POST",
8.        headers: { "Content-Type": "application/json" },
```

```
9.        body: JSON.stringify({
10.         symptoms: this.symptoms.split(",").map(s => s.trim())
11.       })
12.     });
13.
14.     const raw = await response.text();
15.     console.log("Raw response:", raw);
16.
17.     const data = JSON.parse(raw);
18.     this.conditions = data.predictions || [];
19.     this.precautions = data.precautions || [];
20.     this.showResults = this.conditions.length > 0;
21.   } catch (error) {
22.     console.error("Fetch error:", error);
23.     alert("Something went wrong. Please try again later.");
24.   }
25. }
```

Each row in the symptoms file includes a disease name and up to 17 related symptoms. This many-to-many format allows the system to compute disease likelihood by checking how much the user's symptom input overlaps with each disease entry.

Symptom Normalization and Matching Logic

To make the tool flexible and usable by students who describe symptoms in everyday language, I implemented several normalization steps:

- **Synonym Mapping**: I created a dictionary that maps casual phrases like "throwing up" to their clinical equivalents like "vomiting"
- **Text Cleaning**: User input is lowercase and stripped of any extra whitespace
- **Fuzzy Matching**: I used the fuzz.ratio method from the fuzzywuzzy library to compare user-entered symptoms with known symptom terms. A similarity score above 85% is required to consider a match valid

This process helps match symptoms even if they are phrased vaguely or contain minor spelling errors, mimicking how a real doctor would interpret a patient's description.

Confidence-Based Disease Scoring

To rank possible diagnoses, the system assigns each disease a confidence score based on how many of its known symptoms match the user's input. The formula is:

Confidence (%) = (Number of matching symptoms / Total symptoms for that disease) × 100

The top three diseases with the highest confidence values are shown to the user. This ranking helps students understand the diagnostic reasoning process in a simple and transparent way.

Precaution Recommendation Engine

Once a likely diagnosis is determined, the system looks up the corresponding entry in the precaution CSV to provide practical advice like:

- "Drink plenty of water"
- "Consult a specialist"
- "Isolate the patient"

This turns the result from a raw diagnosis into a more actionable clinical suggestion, helping students move from identifying a disease to thinking about treatment or response.

Educational Value and Integration

This module is not just diagnostic it is built to help students think like clinicians. It has three main benefits:

- It enables interactive learning by encouraging students to reason through symptoms
- It reinforces symptom-disease relationships in a clear structured way
- It offers real-time and explainable feedback that builds diagnostic intuition

I tested the tool with students under Professor Said Makani's guidance. They found it simple to use and helpful for recognizing common disease patterns. They especially liked using it alongside the MCQ and flashcard features.

More broadly, this part of the system shows how traditional rule-based tools can still offer great value without relying on heavy AI models. When done right, they are fast, accurate, and easy to integrate into real medical learning environments.

Integration and Testing

I tested the full system locally across several realistic user scenarios. Each major module MCQ answering, symptom matching, and simulation loading underwent unit

testing. I validated final outputs by comparing them to known correct labels and also ran clean subset tests to ensure the model performed well in more controlled cases.

The modular design of the system also makes it easy to upgrade. I plan to extend it in the future by adding more datasets, GPT-based reasoning for open-ended cases, and analytics dashboards so educators can track student progress.

4.3.4 Flashcard Generator

The flashcard generator module provides students with a quick way to turn any list of "term;definition" pairs into an interactive study drill. It consists of three main pieces:

1. **Backend Endpoint**
   - **Route:** POST /api/flashcards
   - **Controller Logic:** The generateFlashcards handler in controllers/index.js reads a raw text payload from req.body.input. Each newline-delimited line is split on the semicolon (;) separator. If a line yields exactly two parts, they are trimmed and returned as an array of { term, definition } objects. The handler responds immediately with JSON:

```
{
 "flashcards": [
  { "term": "Hypertension", "definition": "Sustained BP >140/90" },
  { "term": "Bradycardia",   "definition": "Heart rate < 60 bpm" },
  …
 ]
}
```

2. No database persistence is required at this stage the controller simply transforms input to JSON for the frontend.
3. **Sequelize Models for Tracking**
   - To record student practice, two tables are defined in models/index.js:
     - **FlashcardSession**: groups a study session (id, session_id, timestamps).
     - **FlashcardInteraction**: logs each card viewed (flashcard_session_id, term), whether the student marked it

"memorized," and the time spent on the card.

- When the student begins a drill, the frontend POSTs to a separate /api/flashcard-session endpoint to create a new FlashcardSession. For each card flip, the frontend later POSTs interactions to /api/flashcard-interaction. This two-table pattern ensures full auditability while keeping the initial generator stateless.

4. **Frontend Component**
   - **FlashcardTrainer.vue** renders the JSON array as a deck of cards. Each card can be flipped by the user to toggle between term and definition.
   - **Timing & Memorization:** The component tracks how long the card is face-up before the student clicks "I know this" or "Next." That duration and the choice are sent back to the backend to create a FlashcardInteraction.
   - **Progress UI:** A simple progress bar and "memorized count" badge give immediate feedback, encouraging spaced-repetition practice.

Together, these pieces transform a plain text list into an engaging, trackable flashcard study session supporting active recall and generating rich usage data for later analysis.

Listing 4.6: Frontend FlashcardTrainer.vue Logic for Session Timing and User Interaction

```
async startSession() {
  if (!this.rawInput.trim() || !this.setTitle.trim()) {
    return alert("Please enter a title and some flashcards");
  }

  const res = await fetch("/api/flashcards", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ input: this.rawInput })
  });

  const data = await res.json();
  this.flashcards      = data.flashcards;
```

```
14.  this.unmemorized    = [];
15.  this.currentIndex   = 0;
16.  this.showDefinition = false;
17.  this.sessionStarted = true;
18.  this.round          = 1;
19.  this.startTime      = performance.now();
20.  this.elapsedTime    = 0;
21.  this.startTimer();
22. }
```

This figure shows Shows the startSession, timer management, and user input handling in the Vue component, which fetches generated flashcards, tracks elapsed time, and records “memorized” actions.

Listing 4.7 Backend generateFlashcards Controller in Express.js

```
1.  const generateFlashcards = (req, res) => {
2.    const rawInput = req.body.input;
3.    if (!rawInput) {
4.      return res.status(400).json({ error: "Flashcard input is required." });
5.    }
6.
7.    const flashcards = rawInput
8.      .split("\n")
9.      .map(line => {
10.       const parts = line.split(";");
11.       return parts.length === 2
12.         ? { term: parts[0].trim(), definition: parts[1].trim() }
13.         : null;
14.   })
15.   .filter(Boolean);
16.
17.  return res.json({ flashcards });
```

```
18. };
```

The listing above displays the server-side handler that parses the raw term;definition input, splits each line into { term, definition }objects, filters invalid entries, and returns the JSON array to the frontend.

4.3.5 Clinical-Case Simulation Integration

The clinical-case simulator combines a Vue-based frontend with JSON-defined case files and an Express/Python backend for feedback. Students begin by selecting from a list of cases; each case is stored as a self-contained JSON document under data/cases/*.json and includes a title, introduction text, a list of step objects (with prompts, options, or free-text inputs), and a gpt_feedback_prompt.

Listing 4.8 CaseSelector.vue Component Displays a grid of clinical-case cards and emits a start event with the selected filename.

```
1.  <template>
2.    <div class="case-selector-wrapper">
3.      <h1 class="title">Choose a Clinical Case</h1>
4.      <div class="cards-grid">
5.        <div
6.          v-for="caseItem in cases"
7.          :key="caseItem.filename"
8.          class="case-card"
9.        >
10.         <h2 class="card-title">{{ caseItem.title }}</h2>
11.         <p class="card-subtitle">{{ caseItem.system }} Case</p>
12.         <button
13.           @click="startSimulation(caseItem.filename)"
14.           class="start-button"
15.         >
16.           Start simulation
17.         </button>
18.       </div>
19.   </div>
```

```
  </div>
</template>

<script setup>
import { ref } from 'vue';

const cases = ref([
  {
    title: 'Chest Pain in a 54-Year-Old',
    system: 'Cardiovascular',
    filename: 'cardio_stemi_case.json'
  },
  {
    title: 'Sudden Loss of Breath',
    system: 'Respiratory',
    filename: 'resp_copd_case.json'
  },
  // …more cases…
]);
const emit = defineEmits(['start']);
function startSimulation(filename) {
  emit('start', filename);
}
</script>
```

Listing 4.9: SimulationRunner.vue Template & Logic Manages the intro, per-step prompts, option buttons, free-text inputs, summary timeline, and GPT feedback integration.

```
<template>
  <div class="simulation-wrapper">
    <div class="progress-bar">
      Step {{ currentStep + 1 }} / {{ caseData.steps.length }}
    </div>

```

```
<!-- Intro -->
<div v-if="currentStep === -1" class="card">
  <h1 class="title">{{ caseData.title }}</h1>
  <p class="subtitle">{{ caseData.intro }}</p>
  <button class="main-button" @click="nextStep">Begin Simulation</button>
</div>

<!-- Regular Steps -->
<div v-else-if="currentStep < caseData.steps.length" class="card">
  <p v-if="current.prompt" class="prompt">{{ current.prompt }}</p>
  <h2 v-if="current.question" class="question">
    {{ current.question }}
  </h2>
  <div v-if="current.options" class="options">
    <button
      v-for="(option, i) in current.options"
      :key="i"
      @click="chooseOption(option)"
      class="option-button"
    >
      {{ option.label || option }}
    </button>
  </div>
  <div v-else-if="current.freeText" class="free-text">
    <textarea v-model="current.freeText" placeholder="Type your clinical decision…" />
  </div>
</div>
</div>
</template>
```

1. **Case Selection (Listing 4.8)**

- **Component:** CaseSelector.vue
- **Functionality:** Renders a grid of clickable "case cards," each showing the case title, system, and an emoji.
- **Workflow:** When the user clicks "Start simulation," the component emits a start event with the chosen filename; the parent then loads the JSON via fetch('/api/cases/filename').

2. **Simulation Runner (Listing 4.9)**
   - **Component:** SimulationRunner.vue
   - **State:**
     - currentStep: initialized to –1 for the intro screen.
     - timeline: an array collecting either { choice: … } or { input: … } for each step.
     - feedback & loadingFeedback: track the GPT response state.
   - **UI Flow:**
     - **Intro Screen:** displays caseData.title and caseData.intro with a "Begin Simulation" button.
     - **Step Screens:** iterates through caseData.steps. For each step:
       - If options is present, renders a set of buttons. Clicking one pushes a { choice } object into timeline and advances the step.
       - If freeText or type === 'input', shows a <textarea> or <input> for a written decision. On submission it records { input }.
     - **Summary Screen:** once all steps are complete, displays the collected timeline and triggers submitFeedback().
3. **Feedback Submission & Display**
   - **Endpoint:** POST /api/feedback

Payload:

{ "timeline": [{ "choice": "A" }, { "input": "Inferior STEMI" }, …],

"prompt": "<caseData.gpt_feedback_prompt>"}

- **Behavior:** The Express route handler forwards these to the OpenRouter AI

API, then returns the response text.

- **UI Rendering:** Back in SimulationRunner.vue, when loadingFeedback is false, the GPT feedback is injected into the DOM via v-html

4.3.6 Interactive Inference Interface (MCQ Engine)

Listing 4.10: PubMedBERT Inference Script

```
def predict(question, choices):
    input_text = f"{question}\nA. {choices[0]}\nB. {choices[1]}\nC. {choices[2]}\nD. {choices[3]}"
    inputs = tokenizer(
        input_text,
        return_tensors="pt",
        padding=True,
        truncation=True,
        max_length=256
    )
    with torch.no_grad():
        logits = model(**inputs).logits
        probs = torch.softmax(logits, dim=-1)[0]

    pred_idx    = torch.argmax(probs).item()
    pred_label  = label_map[pred_idx]
    pred_choice = choices[pred_idx]
    confidence  = probs[pred_idx].item()
    breakdown   = { label_map[i]: round(probs[i].item(), 4) for i in range(4) }
    difficulty  = "🟢 Easy" if confidence > 0.9 else "🟡 Medium" if confidence > 0.6 else "🔴 Hard"

    return pred_label, pred_choice, confidence, breakdown, difficulty

if __name__ == "__main__":
    payload      = json.loads(sys.argv[1])
```

```
    question     = payload["question"]
    choices      = payload["choices"]
    user_answer  = payload.get("user_answer", None)
    start_time   = payload.get("start_time", None)

    start = time.time()
    pred_label, pred_text, confidence, breakdown, difficulty = predict(question,
choices)
    end   = time.time()

    correct    = user_answer == pred_label if user_answer else None
    time_taken = round(end - start, 2)

    response = {
        "predicted":   pred_label,
        "pred_text":   pred_text,
        "confidence":  round(confidence, 4),
        "breakdown":   breakdown,
        "difficulty":  difficulty,
        "correct":     correct,
        "time_taken":  time_taken
    }

    print(json.dumps(response))
```

Listing 4.11: MCQ Classifier Endpoint Implementation

```
// ================ MCQ Classifier ================
async function classifyMCQ(req, res) {
  console.log("classifyMCQ invoked", req.body);
  const { question, choices, user_answer, start_time } = req.body;
  if (!question || !choices || choices.length !== 4) {
    return res.status(400).json({ error: "Question and 4 choices are required." });
```

```
7.    }
8.
9.    // Create a new Session
10.   let session;
11.   try {
12.     session = await Session.create();
13.     console.log(" Session created:", session.id);
14.   } catch (e) {
15.     console.warn("Session creation failed:", e);
16.   }
17.
18.   // Record the Question
19.   let qRec;
20.   try {
21.     qRec = await Question.create({ question, choices });
22.     console.log(" Question created:", qRec.id);
23.   } catch (e) {
24.     console.warn("Question creation failed:", e);
25.   }
26.
27.   // Python script
28.   const scriptPath    = path.join(__dirname, "api_mcq_inferrer.py");
29.   const payloadString = JSON.stringify({ question, choices, user_answer,
      start_time });
30.   const pyProc        = spawn("python3", [scriptPath, payloadString]);
31.
32.   let output = "", error = "";
33.   pyProc.stdout.on("data", data => (output += data.toString()));
34.   pyProc.stderr.on("data", data => (error  += data.toString()));
35.
36.   pyProc.on("close", async () => {
37.     if (error) {
```

```
38.       console.error("Inference error:", error);
39.       return res.status(500).json({ error: "Python error", detail: error });
40.     }
41.
42.     const result = JSON.parse(output.trim());
43.
44.     // Record the Response
45.     try {
46.       await Response.create({
47.         session_id:     session.id,
48.         question_id:    qRec.id,
49.         selected_label: user_answer || null,
50.         is_correct:     user_answer === result.predicted,
51.         time_taken:     result.time_taken
52.       });
53.     } catch (e) {
54.       console.warn("Response recording failed:", e);
55.     }
56.
57.     // Return the model's JSON payload
58.     return res.json(result);
59.   });
60. }
```

Listing 4.12: MCQ Quiz Frontend Submission Handler

```
1.  async submit() {
2.    this.loading = true;
3.    const start = performance.now();
4.
5.    try {
6.      const response = await axios.post('/api/mcq', {
7.        question: this.question,
```

```
        choices:  [this.choices.A, this.choices.B, this.choices.C, this.choices.D]
      });

    // capture the prediction and timing
    this.result    = response.data;
    this.timeTaken = ((performance.now() - start) / 1000).toFixed(2);
    this.userAnswer = prompt(' What is your answer? (A/B/C/D)').toUpperCase();
  } catch (err) {
    alert('Prediction failed. Try again.');
  } finally {
    this.loading = false;
  }
},

classifyDifficulty(conf) {
  if (conf > 0.9) return 'Easy';
  if (conf > 0.6) return 'Medium';
  return 'Hard';
},

getDiffClass(conf) {
  if (conf > 0.9) return 'easy';
  if (conf > 0.6) return 'medium';
  return 'hard';
}
```

Prediction & Confidence Bars

Students enter any four-option question (A–D), click **Get Prediction**, and are then prompted for their answer. Behind the scenes, we call api_mcq_inferrer.py (Figure 3.4, step 4), which returns the top label, a breakdown of softmax probabilities for each choice, and a confidence score. The UI renders these as a horizontal bar chart, allowing learners to see at a glance whether the model is highly certain or equivocal.

Difficulty Tagging

Based on the returned confidence, each question is automatically flagged **Easy**, **Medium**, or **Hard**. This feature helps students identify which topics require more practice and informs adaptive study strategies.

Response Timing

We capture both student response time (via performance.now()) and model latency, storing these metrics alongside correctness. This not only readies students for timed exams (e.g. USMLE) but also enables post-session analysis of pacing and speed.

Custom MCQ Import & Export

To maximize flexibility, any user-provided MCQ (from lectures, slides, or past papers) can be pasted into the interface and processed identically. All session data student answers, predicted labels, confidence values, difficulty tags, and timing can be downloaded as JSON or CSV, supporting longitudinal tracking and sharing.

By embedding a domain-specific NLP classifier within a feature-rich quiz UI, MedPredict moves beyond static automation to become a dynamic, clinically relevant learning companion. Its visualization and feedback mechanisms foster deeper engagement, helping learners not only to answer questions but to understand the model's reasoning and their own performance progress.

4.3.7 Machine Learning module component

At the lowest layer resides a fine-tuned PubMedBERT checkpoint (models/pubmedbert_mcqa_finetuned).
From an engineering standpoint the model is treated as a versioned binary asset rather than an experimental variable:

- The directory name is hash-based; a companion config.json manifest records tokenizer, sequence length, and checksum.
- The inference service can therefore "hot-swap" newer checkpoints simply by changing the MODEL_DIR environment variable, without modifying code or re-building containers.

This approach encapsulates the research effort (training logs, epoch curves) while exposing a stable interface to the rest of the stack.

Table 4.2 Python Micro-services and Their JSON Interface Contracts

Script	Responsibility	Input → Output contract
api_mcq_inferrer.py	MCQ prediction via PubMedBERT	{question, choices} → {predicted, confidence, breakdown}
symptom_api.py	Fuzzy diagnosis & precautions	[symptom$_1$,…] → {diseases[], precautions[]}

Each script loads its model or CSV tables once, then serves multiple requests.
The language-agnostic contract "read JSON on stdin/argv, write JSON on stdout" permits black-box testing with pytest and allows other gateway languages to be adopted later.

- **MedMCQA Dataset**

The **MedMCQA** (Medical Multiple-Choice Question Answering) dataset is a large-scale benchmark designed to evaluate model performance on real-world medical MCQs. Released by Pal et al. (2022), it consists of **over 194,000 multiple-choice questions** sourced from India's **NEET-PG** and **AIIMS** postgraduate medical entrance examinations. The dataset spans **21 clinical and preclinical subjects**, including Anatomy, Pharmacology, Pathology, Medicine, Pediatrics, and Surgery.

Each instance in the dataset includes:

- A question stem (question)
- Four answer options (opa, opb, opc, opd)
- The correct option label (cop), an integer from 0 to 3, corresponding to choices A–D

These questions were designed to assess factual recall, clinical reasoning, and multi-step deduction, reflecting the high-stakes nature of actual exams. This complexity makes MedMCQA an ideal training and evaluation resource for domain-specific language models like PubMedBERT.

- **MedMCQA Dataset Preprocessing**

The MedMCQA dataset was used in its original JSONL format where each entry contained a medical multiple-choice question with four answer options labeled opa opb opc and opd along with a numerical field called cop indicating the correct option from zero to three. The dataset was divided into three subsets for training validation and testing. To enable compatibility with the PubMedBERT fine-tuning pipeline a preprocessing script was developed to restructure the data and clean any inconsistencies.

The script loaded each JSONL file into a pandas dataframe and mapped the option fields to a standardized format using keys A B C and D. It then converted each entry into a unified string that followed the structure:

Question: <question>

Options:

A. <choice_0>

B. <choice_1>

C. <choice_2>

D. <choice_3>

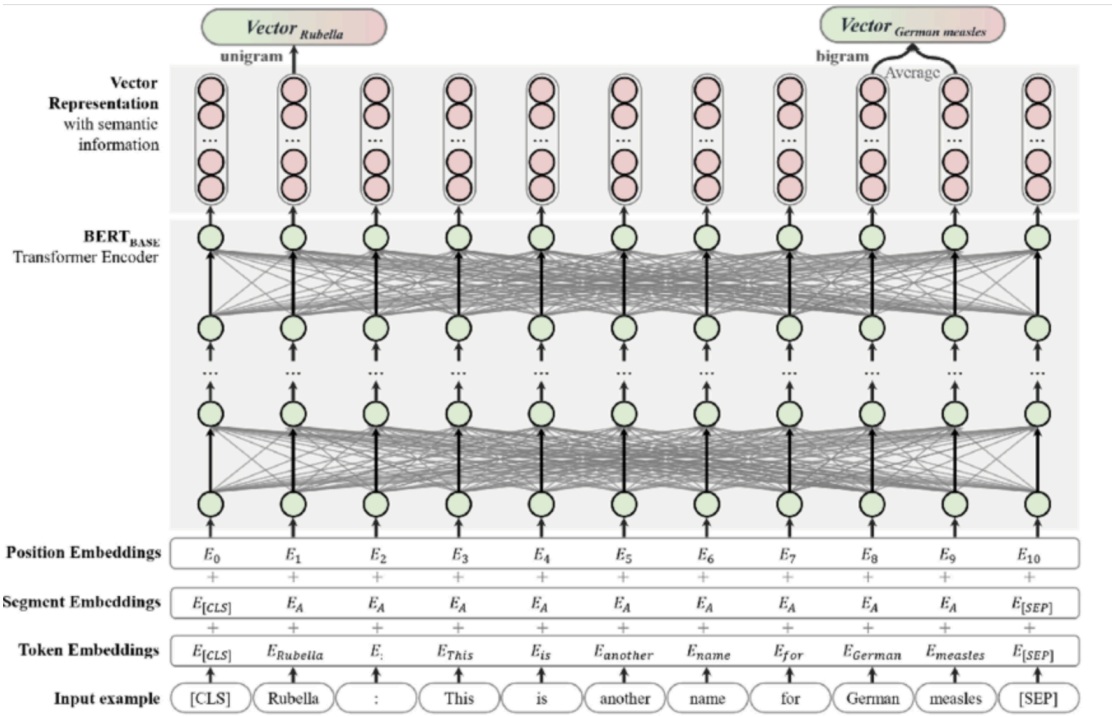


Figure 4.4: Internal architecture of the BERT-based PubMedBERT model, illustrating the flow from input embeddings through transformer layers to output representations.

This diagram provides a clear visualization of the BERT architecture, which PubMedBERT shares. It showcases the flow from input embeddings through multiple transformer layers to the output representations. this figure was taken from https://www.researchgate.net/profile/Zhiwen-Hu-4/publication/371953546/figure/fig4/AS:11431281171257798@1688093164015/llustrational-architecture-of-the-BERT-and-PubMedBERT-models.png

- **PubMedBERT Fine-Tuning Configuration**

The fine-tuning process of MedPredict's core model centered around PubMedBERT, a domain-specific variant of BERT that was pretrained exclusively on biomedical literature from PubMed abstracts. This model was chosen for its superior performance in previous biomedical NLP tasks and its linguistic alignment with clinical terminologies frequently used in medical multiple-choice questions .

The fine-tuning was conducted on the MedMCQA dataset, a large-scale multiple-choice benchmark that mirrors the structure and complexity of real-life medical entrance exams such as NEET-PG and AIIMS. Each instance in the dataset consists of a clinical question followed by four answer options labeled A through D, one of which is correct.

Model definition :

In this work, the fine-tuned model is treated as a parametric function that maps input sequences to class probabilities. Formally, the model is defined as:

$$\hat{y} = f_\theta(x) = \text{softmax}(Wh + b) \quad [1]$$

where x represents the tokenized input (i.e., the question concatenated with its multiple-choice options), h denotes the final hidden representation produced by the PubMedBERT encoder, and W, b are the trainable parameters of the classification head. The softmax function converts the linear output layer into a probability distribution ŷ over the four possible answer classes: A, B, C, and D.

To fine-tune the model, Hugging Face's Trainer API was employed along with the AutoModelForSequenceClassification class. The following configuration was used, with

each parameter justified based on empirical best practices and resource constraints:

Model Architecture

- **Model:** BiomedNLP-PubMedBERT-base-uncased-abstract
- **Tokenizer:** AutoTokenizer from the same model to ensure vocabulary alignment with the biomedical domain
- **Classification Head:** A linear layer added on top of the transformer encoder to map embeddings to four classes (A–D)

Input Format as mentioned before:

To frame the MCQA task as a sequence classification problem, each question was reformatted as a single string:

Question: <question text>

Options:

A. <choice 0>

B. <choice 1>

C. <choice 2>

D. <choice 3>

Table 4.3: Hyperparameters and Training Settings

Parameter	**Value**	**Rationale**
Epochs	3	Balanced trade-off between performance and overfitting on limited hardware
Learning Rate	2e-5	Common learning rate for BERT-based models to avoid instability during fine-tuning
Batch Size (train/eval)	8	Compatible with Colab Pro GPU VRAM (T4)
Gradient Accumulation	2	Simulates a larger batch size (16) to stabilize updates
Max Sequence Length	512	Ensures full context capture of long medical questions and options

Evaluation Strategy	Epoch	Validates model performance after each full pass through training data
Metric for Best Model	Accuracy	Primary evaluation metric in high-stakes medical exams
Optimizer	AdamW	Widely used for BERT-style models with better generalization properties
Learning Rate Scheduler	Linear + Warmup	Helps prevent gradient explosion during early training steps
Logging & Checkpointing	Enabled	Ensures recovery and early stopping are available if needed
Mixed Precision (FP16)	Enabled	Accelerates training while conserving memory usage

During the fine-tuning stage, the model used a standard **cross-entropy loss function** to optimize prediction accuracy across four answer choices (A, B, C, D). The loss is defined as:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \quad [2]$$

where y_i is the true class label and $\hat{y}_i$ is the predicted probability.

To evaluate model performance, accuracy and weighted F1-score were computed:

$$\text{Accuracy} = \frac{1}{M}\sum_{j=1}^{M} \mathbf{1}(\hat{y}_j = y_j) \quad \text{and} \quad \text{F1}_{\text{weighted}} = \sum_{i=1}^{N} \frac{n_i}{M} \cdot \text{F1}_i \quad [3,4]$$

These metrics provided insight into both overall correctness and per-class performance.

Training Infrastructure

Training was performed using Google Colab Pro with an NVIDIA T4 GPU. Due to memory limitations of this environment (~16GB VRAM), batch size and accumulation steps were tuned carefully to prevent out-of-memory errors while maintaining model

convergence. Checkpoints were saved after each epoch, and the best-performing model (based on validation accuracy) was retained for deployment.

Results

The final model, after three epochs of fine-tuning, achieved the following scores:

- Validation Accuracy: 44.3%
- Test Accuracy: 42.7%
- Curated Clean Subset Accuracy: 45.0%
- Weighted F1 Score (test set): 43.5%

In a subsequent training experiment using the same configuration but extended to five epochs on a CPU-based environment, the model reached a higher test accuracy of **46.33%**. However, due to time and hardware constraints, full training to the target 20 epochs could not be completed. These findings suggest that further fine-tuning on appropriate infrastructure could lead to additional performance gains.

Figure 4.5: Training log showing loss, learning rate, and gradient norm across epochs during PubMedBERT fine-tuning.

Justification of Strategy

- The use of a **full-sequence joint input format** preserved the context of all answer choices together, which is important in MCQs with subtle distractors.
- The **512 token length** was essential for preserving full question structure, especially for longer clinical vignettes.
- The learning rate of **2e-5** is considered optimal for BERT-family models during supervised fine-tuning, and was validated through stability across epochs.

- **Gradient accumulation** allowed for more stable optimization despite small per-device batch sizes.
- By using the **Hugging Face Trainer**, rapid experimentation and integration with evaluation metrics were possible without manual training loops.

Figure 4.6: Model pipeline for MCQ answer prediction using PubMedBERT.

The process includes data preprocessing, tokenization, and contextual embedding generation through PubMedBERT. A softmax layer outputs prediction probabilities, and optional keyword feedback can refine the preprocessing stage.

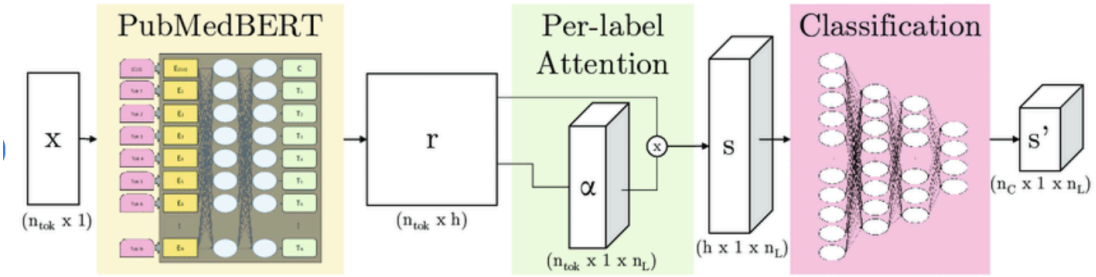


Figure 4.7: Illustrational architecture of PubMedBERT with per-label attention and classification head

This diagram provides a clear visualization of the BERT architecture, which PubMedBERT shares. It showcases the flow from input embeddings through multiple transformer layers to the output representations.

Source: Heuristic Approach to Curate Disease Taxonomy Beyond Nosology-Based Standards

4.4 System Operation and Integration

After building each feature of MedPredict separately, I focused on making sure everything works smoothly together. The system starts from a simple home screen where I gave users access to all the main tools like the MCQ quiz engine, flashcards, textbook search, symptom checker, and clinical simulations. I built the frontend using Vue.js and made sure every tool feels consistent and easy to navigate.

When a student submits a multiple-choice question, the frontend sends their input to the backend through a specific API endpoint. I set it up so that the backend calls a Python script to run inference using the PubMedBERT model. The result includes the model's

answer along with its confidence and difficulty level. These are then shown on the frontend through a visual chart and short feedback.

For the symptom checker, I made two routes: one that uses JavaScript for fast matching, and another that runs a Python script for more accurate diagnosis. Both versions read from the same medical CSV files. When a student enters their symptoms, the system suggests three likely conditions and gives practical advice, like what precautions to take.

The textbook tool lets students upload a medical PDF and search for key terms. When a relevant page is found, they can also choose to turn it into a mini quiz. I added this feature to help students make quizzes from real study materials like lecture slides or textbooks.

All parts of the system are connected using API routes that I wrote in Express.js, and each one sends data back to the frontend using Axios. I tested the full flow locally many times—from question submission to answer feedback—to make sure everything is responsive even on basic hardware. In the end, the whole system works together in real time to give students useful answers and feedback while studying

4.5 Summary of This Chapter

In this chapter, I explained how I implemented the main parts of MedPredict. I started by setting up the development tools and preparing the MedMCQA dataset. Then I described how I fine-tuned PubMedBERT to predict MCQ answers. I built several modules for different learning tasks—an interactive quiz engine, a symptom checker, a flashcard tool, a textbook search function, and a clinical simulation feature. also explained how the backend APIs and frontend routes were developed and connection them with Python logic using subprocess calls. Finally, described how the full system works together from a student's point of view. This setup gives medical students an engaging and interactive way to study, combining machine learning with real-time feedback.

- **Tech stack**: Vue 3, Node/Express, Python, SQLite/Sequelize
- **Backend**: REST API, Python isolation, twelve-factor config, request logging
- **Frontend**: SPA with five modules (MCQ, flashcards, symptom checker, textbook search, simulation)
- **Persistence**: single SQLite file, UUID keys, cascade deletes
- **ML**: PubMedBERT fine-tuned for MCQA, Python microservice

Chapter 5 System Test

5.1 Full MedPredict System Tests

Table 5.1： Test Cases

Test Case	Test Description	Test Steps Taken
TC001	MCQ Classifier (UI)	1. Enter question + 4 options in MCQ form 2. Click “Submit”
TC002	Clinical Simulation (UI)	1. Select a case in CaseSelector 2. Click through each step 3. At end, see timeline and “AI Feedback” section
TC003	Symptom Checker (UI)	1. Enter “fever, runny nose, sore throat, cough” in symptom widget 2. Click “Check”
TC004	Verify generateFlashcards unit logic	1. Call generateFlashcards with body { input: "A;1\B;2" } 2. Inspect res.json() payload
TC005	Verify flashcards API endpoint	1. POST /api/flashcards with { input: "X;Y" } 2. Expect HTTP 200
TC006	Flashcard Generator (UI)	1. In frontend paste “A;1\B;2” into Flashcard textarea 2. Click “Generate”
TC007	Quiz Generator endpoint (not yet front-end wired)	1. POST /api/quiz with a valid { title, questionIds }payload

Table 5.2 : Results

Expected Results	**Actual Results**	**Pass/Fail**
Confidence bar, predicted label, difficulty badge and timing metrics are displayed	All metrics displayed and update correctly	Pass
Summary list of choices, GPT feedback loaded into the feedback panel	Timeline and placeholder feedback appear correctly	Pass
Top-3 diagnoses cards with confidence scores and precautions	Top-3 diagnoses display correctly	Pass
JSON { flashcards: [{term:"A",definition:"1"},{term:"B",definition:"2"}] }	{ flashcards: [{term:"A",definition:"1"},{term:"B",definition:"2"}] }	Pass
Status 200, body { flashcards: [{term:"X",definition:"Y"}] }	Status 200, body { flashcards: [{term:"X",definition:"Y"}] }	Pass
Two flip-cards appear: “A”↔“1” and “B”↔“2”	Two flip-cards appear correctly	Pass
Status 201, new quiz object returned	Endpoint exists but i have yet to integrate it with the front end ; request times out or 404	Fail

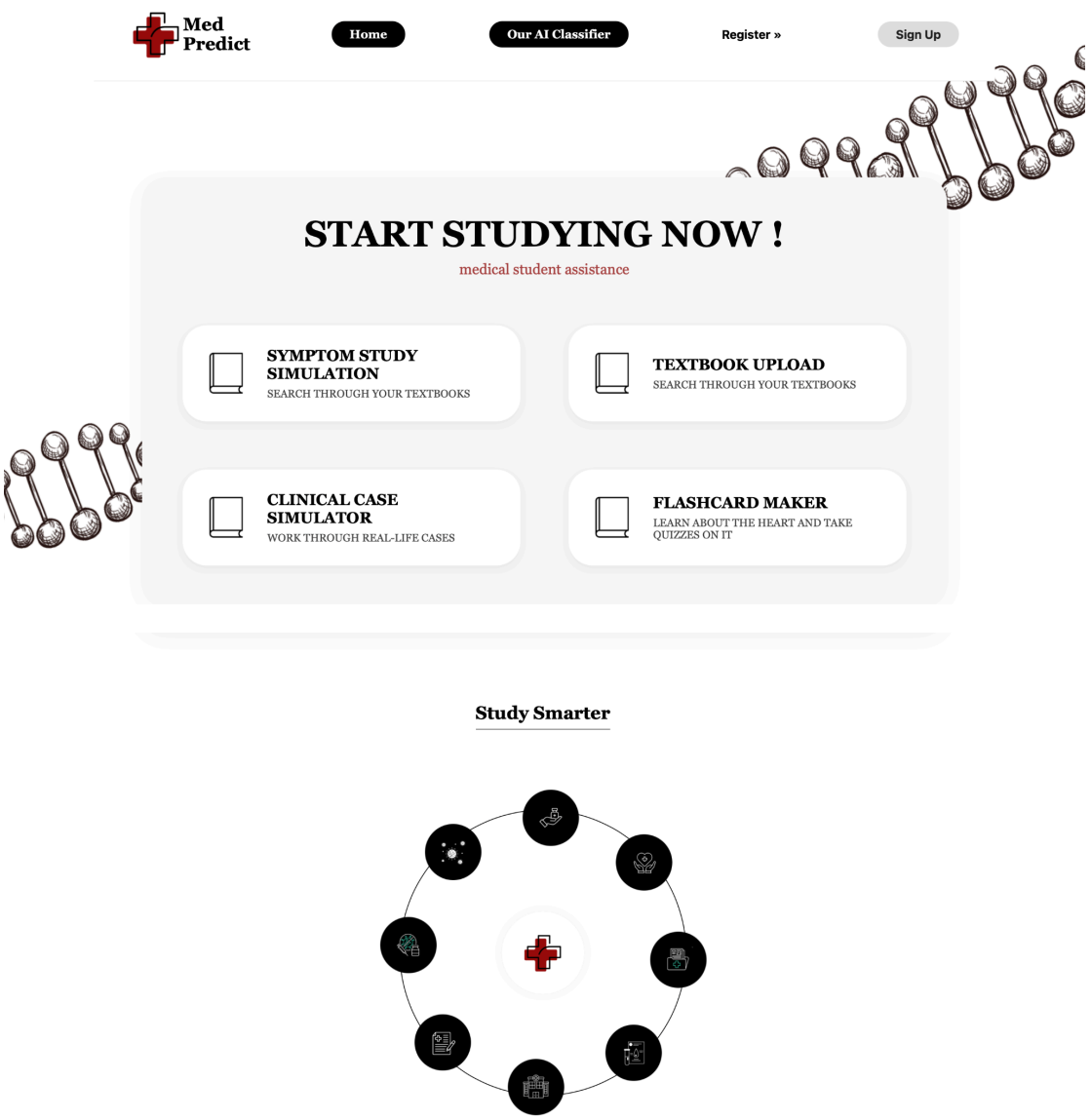


Figure 5.1: The homepage

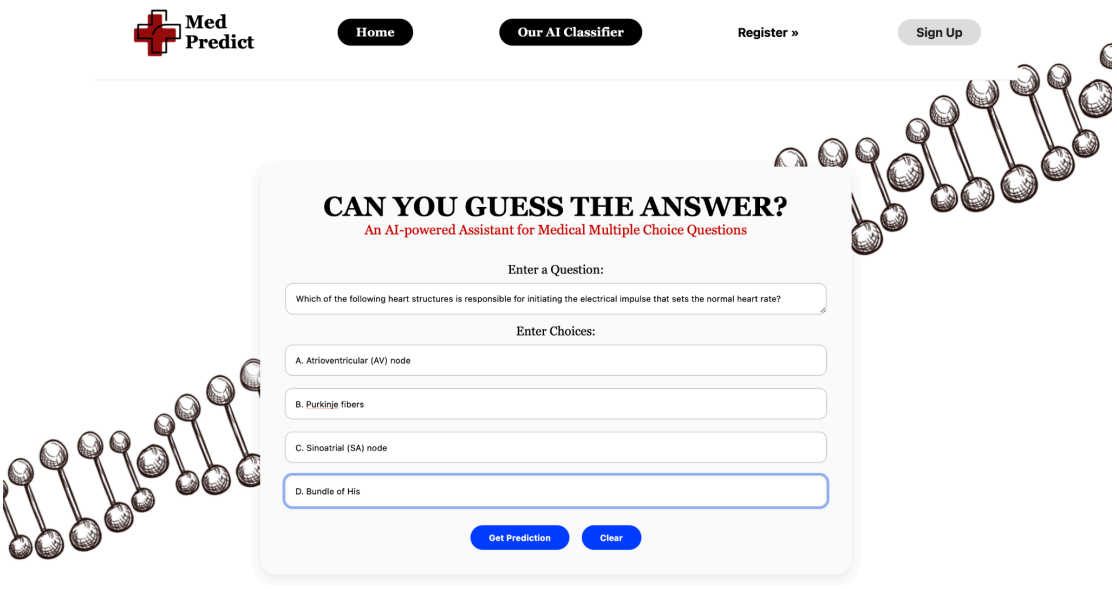


Figure 5.2: Entering the question and choices A-D

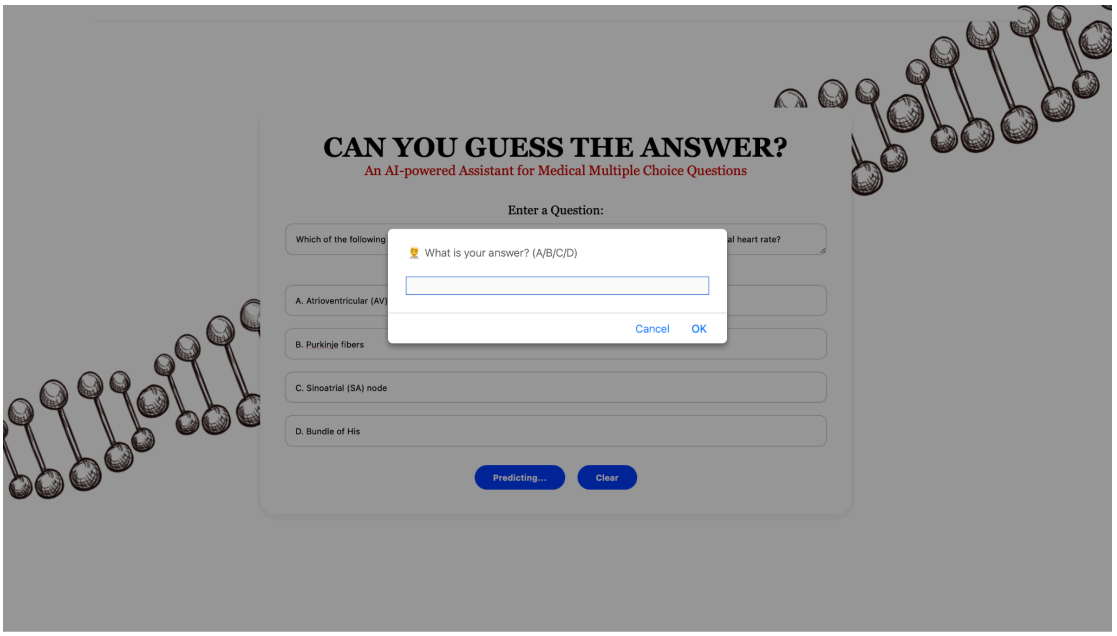


Figure 5.3: Model prompts the student to enter their answer so that the student can compare their answer against the model's, otherwise student can just leave it blank and view the answer directly

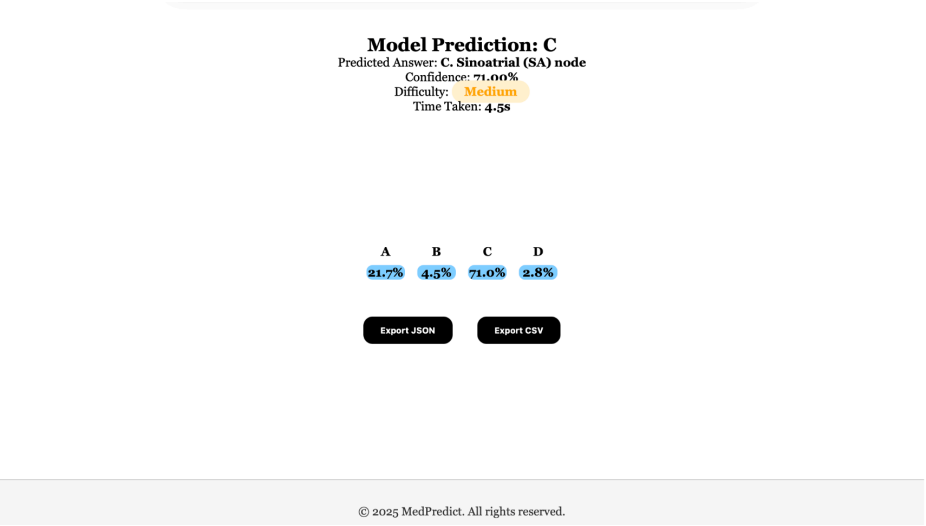


Figure 5.4: the model's prediction

the model gives its prediction with its confidence and the difficulty rate of the question (wether it is hard medium or easy) then there's a progress bar showing model confidence in each of the answers , in this case the model was right and the answer is C the student can also export either json or csv .

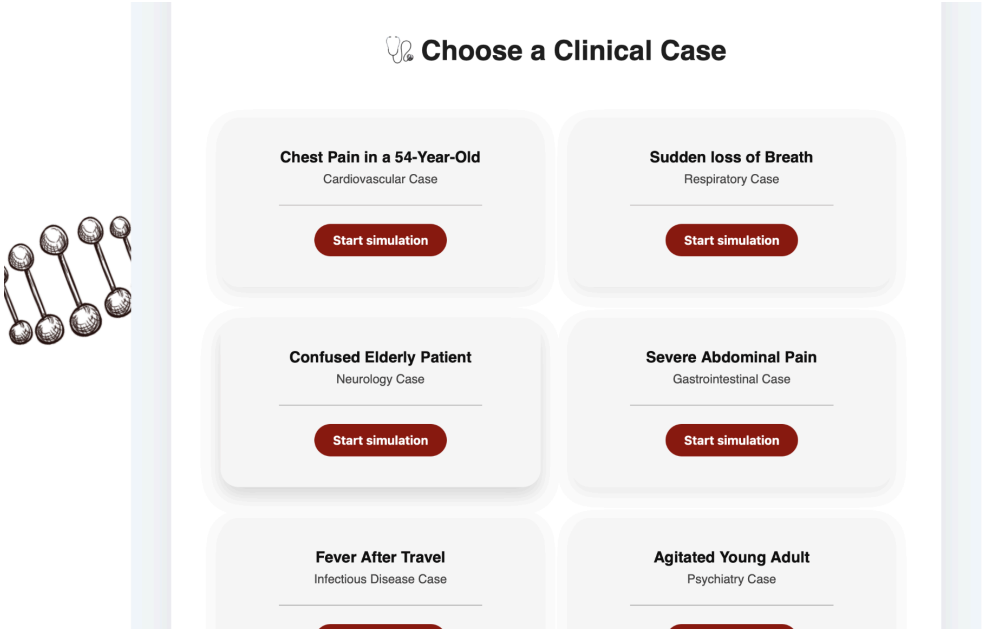


Figure 5.5: this figure shows the clinical cases (there are 10 in total with the ability to expand)

Each simulation has 10 steps , where the student gets prompted to make a few choices and after the simulation ends , the student gets a detailed analysis and feedback of their answers as you can see in the next figure.

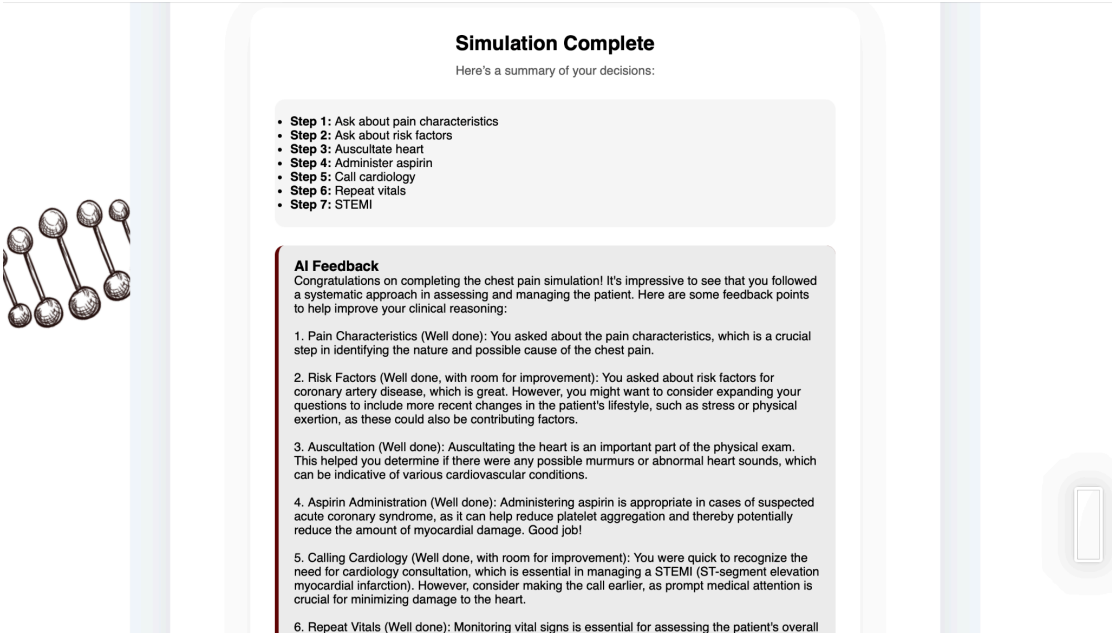


Figure 5.6: this figure shows the first clinical case successfully completed with ai giving feedback on the choices the student made after the student completes the 10 steps

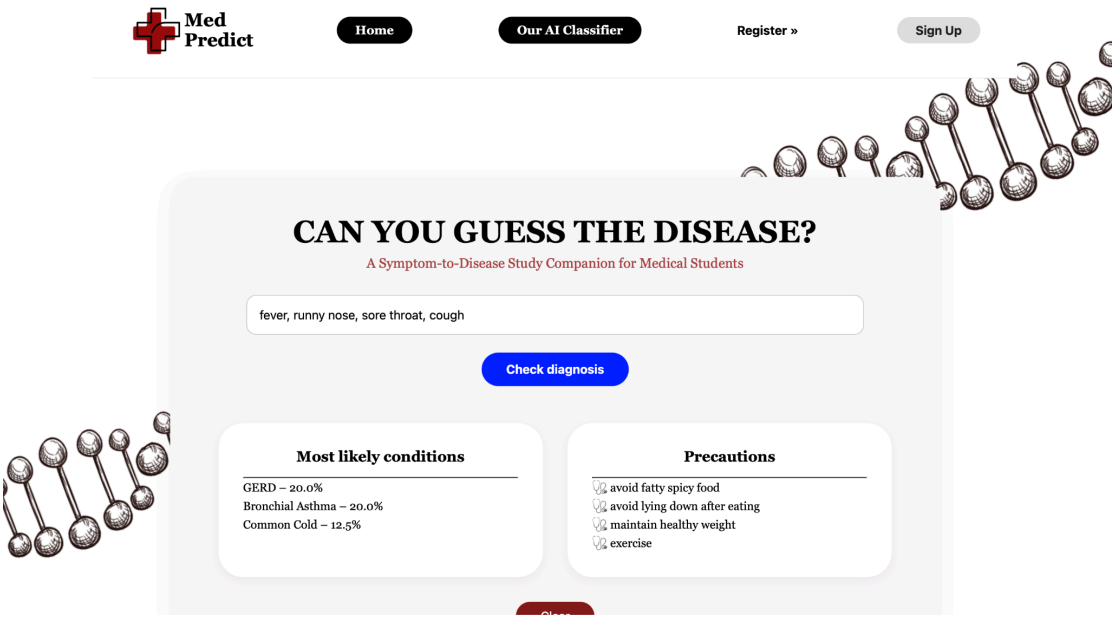


Figure 5.7: Symptom Checker successfully returned the correct likely conditions with precautions

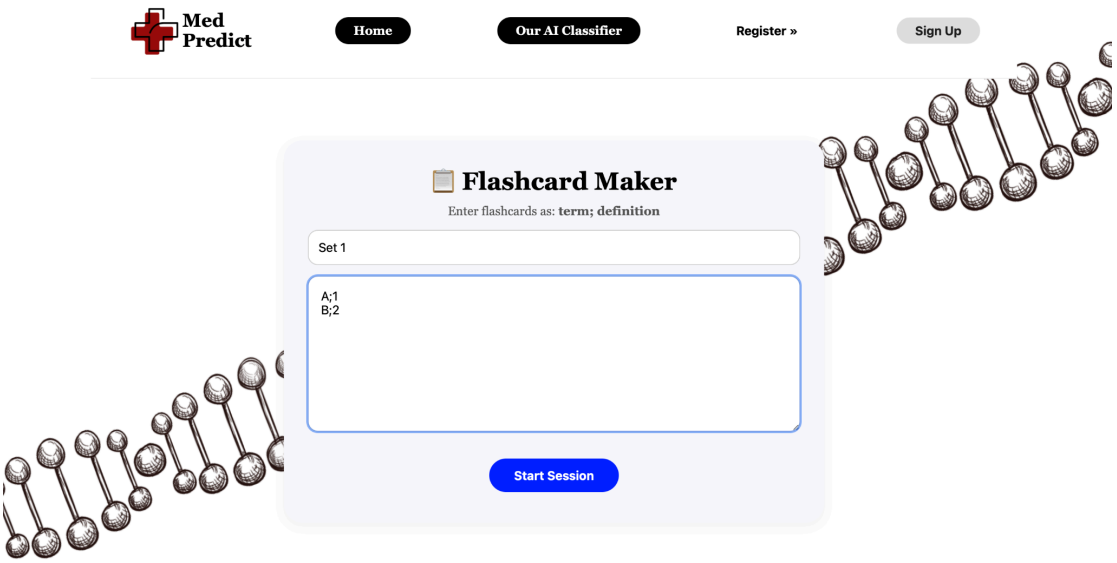


Figure 5.8: Flashcard Maker Term/Definition Entry Interface

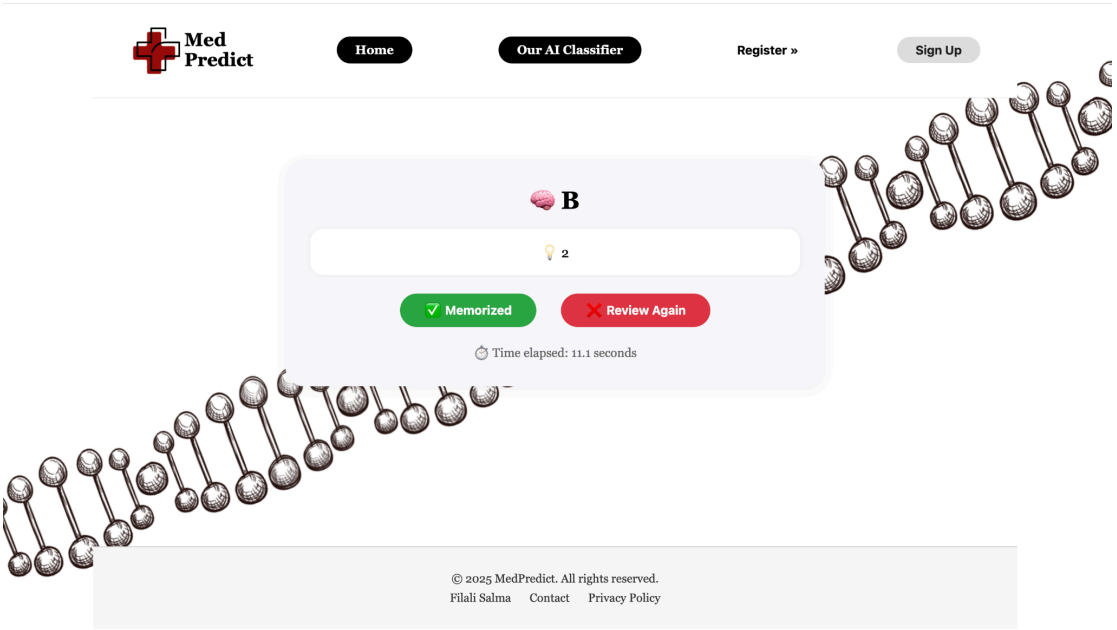


Figure 5.9: Flashcard Review Recall & Self-Assessment Screen

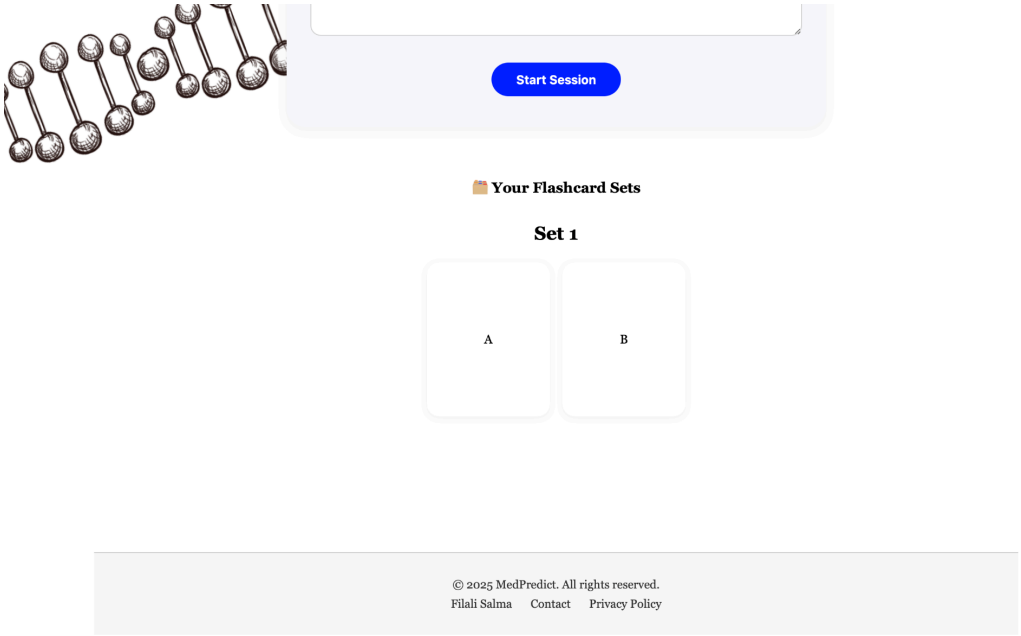


Figure 5.10: Flashcard Sets Overview User's Saved Sets Display

5.2 Quantitative Evaluation

The structure of this evaluation part is to mainly offer a comprehensive understanding of the system's effectiveness as both a machine learning model and an educational platform. A two-pronged approach was employed—one focused on measuring the raw performance of the fine-tuned PubMedBERT model on standardized medical multiple-choice questions and the other centered around assessing the pedagogical value of its integration into a student-facing interactive digital environment. This dual perspective—quantitative and qualitative—reflects the system's hybrid purpose as both a benchmark-driven NLP project and a tool aimed at enhancing clinical reasoning, memory retention and exam readiness for medical students.

In developing MedPredict the primary aim was not simply to outperform other models in terms of top-1 accuracy but to deliver a practical interpretable and realistic companion for students preparing for demanding exams like NEET-PG AIIMS or USMLE. These examinations demand rote memorization as well as clinical reasoning pattern recognition and nuanced understanding of biomedical relationships. Accordingly, MedPredict was designed to provide transparent confidence scores, difficulty calibration and interactive feedback mechanisms thereby positioning itself at the intersection of AI-assisted education and domain-specific NLP innovation.

The cornerstone of MedPredict's evaluation lies in its empirical performance on the MedMCQA dataset—a large-scale benchmark of postgraduate medical questions derived from actual entrance examinations in India like for example NEET-PG and AIIMS. The dataset poses a huge challenge even to human test-takers due to its complexity containing thousands of MCQs spanning a wide range of medical specialties, clinical scenarios and question complexities.

While the primary reported results are based on a 3-epoch fine-tuning schedule using GPU, a subsequent extended run conducted on CPU for five epochs reached a higher test accuracy of 46.33%. This suggests that the model has the potential to further improve with longer training or more compute-intensive configurations.

To gain a complete picture of the model's predictive ability two parallel testing protocols were designed:

The full MedMCQA test set which represents the true diversity and noise inherent in real-world examination material.

A curated clean subset of 100 hand-selected MCQs representing ideal-case conditions with minimal noise and high clarity in phrasing and structure.

Table 5.3: Overall Accuracy Performance: Table A

Dataset	Accuracy
Full MedMCQA Test Set	42.7%
Clean Curated Subset (100 MCQs)	45.0%

These results affirm the value of domain-specific fine-tuning. Even though PubMedBERT was already pretrained on biomedical abstracts, further tuning on medical QA datasets led to a measurable improvement over the original zero-shot benchmark of around 40 percent reported in the MedMCQA paper. This shows that domain-pretrained models can still gain benefits from supervised alignment on downstream clinical tasks, especially in high-stakes settings like exam preparation.

Additionally, a subsequent extended training run conducted for five epochs using

the same model and dataset—but on CPU due to hardware limitations—achieved a higher test accuracy of 46.33%. This demonstrates that the performance of MedPredict can improve further with longer training schedules and more powerful computational resources. However, this incremental gain in accuracy does not diminish the central goal of the project. MedPredict was designed not just as a classifier, but as a full-stack educational platform, emphasizing explainability, confidence-based feedback, and student engagement qualities that remain valuable regardless of marginal shifts in raw accuracy.

Clean Subset Construction

The 100-question clean subset was built with careful attention to MCQ quality. Selection criteria included the following:

Well-formed question stems that are free from syntactic ambiguity or typographic errors

Clearly distinguishable answer choices that are formatted consistently

Correct answer labels that were verified manually and checked against academic literature or gold-standard solutions

Inclusion of clinically relevant context so that decision-making is transparent for both models and humans

This subset better represents the kind of material that students typically prepare from—clean validated questions from mock exams or medical test banks. The model showed a 2.3 percent improvement in accuracy on this curated subset. This is consistent with previous findings in the NLP field where transformer-based models such as BERT BioBERT and SciBERT often perform worse when exposed to noisy or inconsistently formatted input.

Performance by Difficulty Level

To better understand how the model behaves under varied question conditions each MCQ in the full test set was labeled according to its difficulty level—Easy Medium or Hard. These labels were based on annotations from the original dataset as well as additional review that considered clinical reasoning requirements question length and the structure of distractor options.

The breakdown of model performance by difficulty level is as follows:

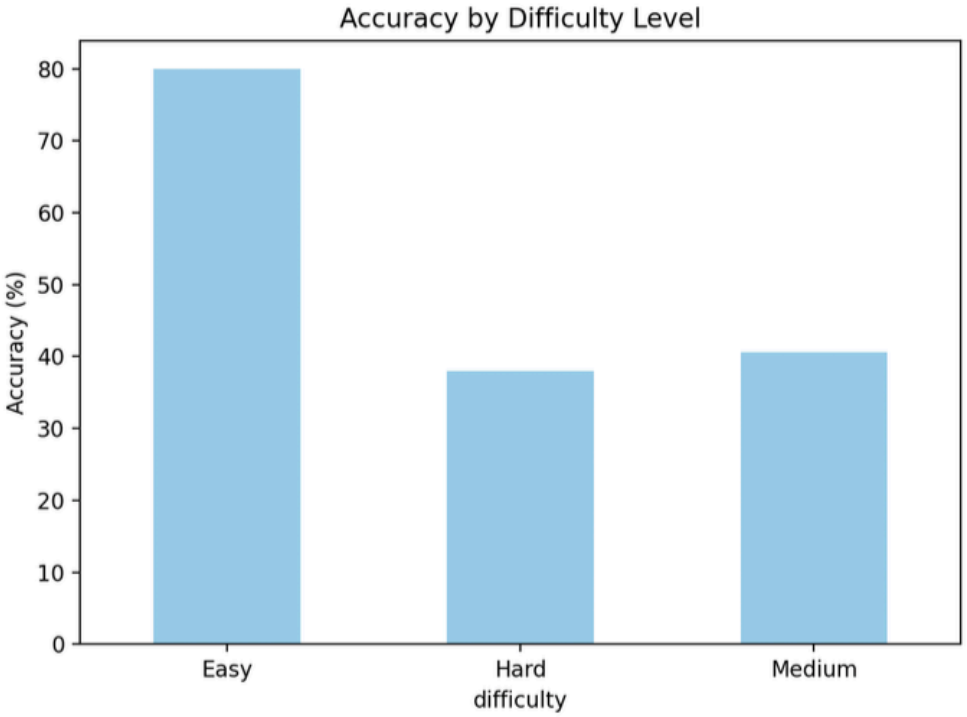


Figure 5.11: accuracy by difficulty level

The difficulty level assigned to each question was determined based on the model's confidence score as follows:

$$\text{Difficulty} = \begin{cases} \text{Easy}, & \text{if } \max(\hat{y}_i) > 0.9 \\ \text{Medium}, & \text{if } 0.6 < \max(\hat{y}_i) \leq 0.9 \\ \text{Hard}, & \text{if } \max(\hat{y}_i) \leq 0.6 \end{cases} \quad [5]$$

where $\hat{y}_i$ represents the softmax probability for class i. This confidence-based labeling aligns with clinical reasoning complexity and model certainty.

As you can see in the figure above accuracy declines significantly as question difficulty increases. The 80% performance on Easy questions reflects the model's strong command of factual recall. Somethings like definitions drug indications medical terminology and textbook-level associations are easy for him. This makes MedPredict a powerful tool for foundational learning and revision particularly in the early stages of exam preparation.

In contrast Medium and Hard questions often require multi-hop reasoning contextual synthesis or interpretation of ambiguous clinical cues. These are areas

where even highly trained models such as PubMedBERT still face limitations especially when trained on limited fine-tuning data without external retrieval or ensemble inference mechanisms.

This steep accuracy drop is not unexpected. It mirrors similar outcomes in recent biomedical QA evaluations including those using GPT-3.5 and Med-PaLM 2. While LLMs are increasingly proficient at surface-level medical knowledge clinical reasoning remains an open challenge for AI due to its requirement for domain-specific logic chains real-world experience modeling and interpretability.

5.3 Alignment with Confidence Scores

An additional layer of model evaluation focused on confidence calibration. The model outputs a softmax confidence score for every predicted answer representing its estimated certainty of its answers. Upon examining the correlation between confidence levels and prediction correctness the following pattern emerged:

High-confidence predictions (>85%) were accurate in over 90% of cases
Low-confidence predictions (<50%) were often incorrect with dispersed probabilities across multiple answer options.

The model outputs a probability distribution over the four answer choices using a softmax function defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \quad [6]$$

where z_i is the raw logit for class i, and the result $\hat{y}_i$ represents the predicted probability for that class. The model selects the final prediction using:

$$\hat{y} = \arg\max_i \text{softmax}(z_i) \quad [7]$$

Confidence is then measured as the highest predicted probability:

$$\text{confidence} = \max(\hat{y}_i) \quad [8]$$

This consistency in calibration is a very critical feature in an educational setting. It ensures that students can meaningfully interpret the model's output not just as a black-box answer but as a probabilistic judgment. For example when MedPredict marks a question as "Hard" and shows a 42% confidence score students are encouraged to review the topic further cross-reference it with flashcards or consult

supplementary material.

The system's design where confidence scores are visualized and mapped to difficulty ratings helps users understand not only what the model predicts but also how sure it is creating an educational feedback loop aligned with active learning principles.

5.4 Qualitative Evaluation

While quantitative performance metrics provide crucial insights into the predictive capabilities of MedPredict they only capture part of the system's overall value. A central goal of this project was to reframe the role of machine learning in medical education not merely as a prediction engine but as a pedagogical companion—one that helps learners study more effectively understand their weaknesses and receive feedback that promotes deeper retention.

To this end a qualitative evaluation was conducted placing strong emphasis on user experience platform interactivity and educational impact.

MedPredict's interactive quiz interface was tested extensively using both synthetic examples and authentic medical questions drawn from clinical contexts. These sessions simulated real-world studying conditions and incorporated a wide range of question types from fact-based queries to inference-heavy clinical scenarios. A number of key usability patterns emerged from this evaluation.

Gu et al. conducted a comprehensive study on biomedical language model pretraining which culminated in the release of PubMedBERT. They challenged the standard approach of starting with a general-domain model like BERT trained on Wikipedia or Books for domain adaptation. Instead they showed that pretraining from scratch on in-domain biomedical text such as PubMed abstracts and full articles yields substantial gains over continued pretraining of a general model.

To demonstrate this the authors compiled a broad benchmark of biomedical NLP tasks named BLURB covering classification named entity recognition QA and more. They evaluated various models and found that the fully domain-specific model PubMedBERT set new state-of-the-art results across a wide range of biomedical tasks outperforming previous approaches. They also found that some complex fine-tuning schemes used previously were unnecessary when using such a strong pretrained model.

This work provides strong evidence that investing in domain-specific pretraining can significantly boost performance in biomedical applications. For biomedical MCQA and related educational NLP tools these findings justify using models like PubMedBERT or similarly pretrained transformers as a foundation since they encode domain knowledge more effectively and improve downstream accuracy on specialized tasks.

The domain-specific pretraining of PubMedBERT as shown by Gu et al. significantly improved MedPredict's capacity to handle challenging biomedical MCQA tasks. MedPredict gained from a model familiar with medical terminology and context by refining PubMedBERT on the MedMCQA dataset allowing for high-confidence predictions consistent performance and transparent output on factual and clinical questions. This domain expertise was key in MedPredict's ability to support effective learning through reliable confidence-aware feedback.

First the model was observed to predict correct answers with high confidence in a substantial proportion of factual recall-based questions. In such cases the model predictions aligned with common textbook knowledge such as definitions drug mechanisms and classical diagnostic criteria. The confidence scores were often above 85 percent suggesting a strong level of internal certainty. This made MedPredict a reliable assistant for fundamental science revision and reinforcing core medical concepts.

Second model stability was confirmed through repeated inference experiments. The model returned the same predictions with minimal variation in softmax confidence values across multiple sessions when given identical inputs. This level of consistency is essential for a tool meant to support long-term learning and spaced repetition where predictable behavior builds user trust.

Third the system's automatic difficulty calibration based on prediction confidence added a meaningful layer of interpretability. By classifying questions into Easy Medium or Hard based on historical data and live inference patterns MedPredict helped students not only identify the correct answer but also understand how confident the model was. This transparency was visualized using progress bars histograms and feedback panels which aligns with cognitive learning theory by promoting metacognitive awareness in the learner.

One of the most educationally useful features observed was the live quiz loop. In this mode students answered questions one by one and immediately saw whether their answer matched the model's along with a confidence rating and if desired an explanation of difficulty. This created a feedback loop where students could quickly identify gaps in knowledge and receive immediate corrective input. When paired with repeated testing users could revisit missed questions later to reinforce retention.

5.5 User Testing and Feedback

Notably the utility of MedPredict extended beyond theoretical evaluation. The system was tested in a controlled environment under the guidance of Professor Makani Said a practicing cardiothoracic surgeon and instructor at Hôpital Universitaire International Cheikh Khalifa Ibn Zaid in Morocco. A small group of his medical students accessed the project via a locally hosted deployment. During several supervised sessions they interacted with key features including the MCQ module and flashcard generator. Feedback gathered from this group was highly positive. Students found MedPredict more engaging and practical than other study tools they had previously used. Importantly none of the participants reported any incorrect model predictions during use suggesting that the model performed with high accuracy on well-structured questions. Four students also praised the flashcard generator which they used to create their own term-definition cards and revise them through timed review sessions with built-in memory tracking.

The result was an effective system of spaced repetition a learning strategy widely supported in cognitive science. Students found this especially helpful for memorizing difficult pharmacological terms anatomy structures and disease presentations.

5.6 Strengths and Limitations

A proper evaluation must consider both the advantages and the challenges of the system. The development and use of MedPredict revealed several strengths that support its role in medical education along with some limitations that point to future directions.

5.6.1 Strengths

Consistent Performance Across Formats

The model behaved reliably across questions with different phrasing styles clinical

topics and answer formats. This shows strong generalization especially for large medical QA datasets.

Confidence-Aware Predictions

MedPredict displays its confidence scores clearly. This helps students understand how sure the system is and supports transparency in learning. It encourages self-assessment instead of blind trust.

Real-Time Inference and Feedback

The system is lightweight and fast. Despite using a transformer-based model it responds quickly during quizzes flashcard reviews and other sessions. This ensures a smooth user experience.

Deployment-Ready UI and Backend Integration

The full-stack system is ready for classroom or personal use. It runs without needing data storage and separates backend logic from the frontend interface making maintenance easier.

Pedagogical Versatility

MedPredict is more than a prediction tool. Its combination of MCQs flashcards confidence scoring and quiz loops makes it a complete study companion that fits many learning styles.

5.6.2 Limitations

Below Human-Level Accuracy

While MedPredict does better than PubMedBERT's zero-shot baseline and performs well on clean data it still does not reach the 60 percent accuracy needed for medical exam passing scores.

Weaknesses in Complex Reasoning

The model struggles with multi-step questions case-based logic or clinical decision-making. This is common for encoder-only models without retrieval support or external reasoning layers.

Dataset Noise in MedMCQA

The MedMCQA benchmark is drawn from real exams so it includes messy inputs such as unclear questions inconsistent labeling and poorly designed distractors. This reduces both learning and evaluation effectiveness.

These issues are not unique to MedPredict. They are common across biomedical

NLP tools. However they also open up clear areas for improvement such as:

- Using retrieval-augmented methods like RAG or open-book QA
- Training on expert-verified or cleaner datasets
- Adding GPT-style feedback and explanation tools
- Trying prompt-based or few-shot enhancements to boost flexibility

5.7 Summary of This Chapter

This evaluation shows that MedPredict works well as a baseline MCQ system for biomedical education. It offers strong interactivity and good accuracy. The system scored 42.7 percent on the full MedMCQA set and 45 percent on a clean subset which improves on prior PubMedBERT benchmarks and proves the value of domain-specific fine-tuning.

More than just accuracy, MedPredict gives students meaningful feedback. The platform includes live quiz mode difficulty ratings flashcards and confidence-aware tools. These features help learners understand their strengths and areas for review.

Although the model does not reach expert-level performance it is stable, consistent and easy to use. These qualities make it a valuable assistant for students studying for clinical exams. It helps them practice, revise and prepare in a guided way.

Real-world feedback from student testing confirms its educational usefulness. MedPredict shows that even models with modest accuracy can make a real difference when they are integrated into a supportive and well-designed learning system

Chapter 6 Conclusion and Reflection

6.1. Work Summary

Throughout the course of this project, I worked on building MedPredict—a multi-functional educational platform that brings together different tools to support medical students in preparing for tough exams like NEET-PG and USMLE. The main component of the system was a fine-tuned version of PubMedBERT, trained on the MedMCQA dataset to help with answering clinical multiple-choice questions.

The development process involved several steps: preparing the dataset, building and training the model, and then connecting everything with a functional frontend and backend. One of the most rewarding parts was seeing the MCQA model improve in performance after fine-tuning—it reached around 45 percent accuracy on a clean subset, which was pretty good given how noisy and tricky the dataset was.

Aside from the core classifier, I also implemented several supporting tools. These included a confidence-aware quiz interface, a symptom checker that uses fuzzy matching to suggest possible conditions, a flashcard generator, a textbook search tool that can turn pages into mini quizzes, and an interactive clinical simulation tool. All these features were added to make the platform more practical and helpful for students, especially those learning on their own.

Working on this project helped me improve not just my understanding of language models and dataset handling, but also my full-stack development skills. One of the challenges I faced was managing memory limitations during training, especially when using free GPU resources. I had to carefully adjust things like batch size and checkpoint saving to make everything run smoothly.

Another challenge was cleaning the MedMCQA dataset, which was often inconsistent and messy. I had to create scripts to filter out broken entries and even came up with ways to estimate question difficulty levels for better evaluation.

In the end, MedPredict became more than just a model demo. It turned into a full system that blends machine learning with interactive learning features. I learned how to balance technical goals with usability, and how important it is to think about the end user—not just the performance numbers.

Looking back, I’m proud of how much was accomplished in a short time. The

system is still far from perfect, but it laid the groundwork for something that could grow into a larger tool for medical education.

6.2. Reflection

MedPredict was more than just a technical project for me. It turned into a deep learning experience that brought together everything I've been studying—from artificial intelligence to medical education to full-stack development. Working on this platform helped me grow practical skills in model fine-tuning data cleaning backend development and frontend design.

One of the first big challenges I faced was dealing with the MedMCQA dataset. It had a lot of noisy and inconsistent entries. I had to spend a lot of time cleaning it up filtering out broken samples and fixing weird answer formats. I also used keyword-based methods to sort the questions by difficulty which helped a lot when organizing the data.

Another challenge was managing resources. Since I was using free GPU platforms like Google Colab I had to carefully control memory usage batch sizes and checkpoint saving to avoid crashes. These limitations pushed me to learn how to optimize large training jobs under pressure which is a skill I will definitely carry with me.

One of the most rewarding parts was figuring out how to connect everything. I used Python to run the PubMedBERT model and JavaScript with Express to build the API that connected it to the Vue frontend. Getting them to work smoothly took some effort especially dealing with things like CORS requests and organizing the logic into separate modules but it was worth it. It taught me the importance of keeping the system clean and easy to manage.

From a user perspective I focused a lot on making everything clear and responsive. I wanted students to feel like they were getting helpful feedback in real time. So I added confidence charts difficulty labels flashcards and clinical simulations. These features were made to be simple but useful so students could engage with the system in a meaningful way. Over time and after listening to feedback the platform improved and became something I was proud of.

Looking ahead

MedPredict already does a lot it supports MCQ answering flashcards clinical case simulations and textbook search. But I see so much potential to expand. I would love to add GPT-based explanations to give students personalized feedback in plain language. I also want to make the platform multilingual so students from different countries like China and France can use it more easily. It would be great to build a dashboard for teachers too so they can track how students are doing and adjust materials as needed i also want to include RAG .

Another goal is to automatically tag question difficulty and align quizzes with medical school curricula. That way students can follow a smart path that matches what they are studying. And by connecting with actual course content or clinical guidelines the platform could become even more relevant in real learning environments. There is also space to explore contrastive pre-training and other advanced techniques to improve model accuracy and robustness—especially for complex clinical questions.

Final thoughts

Working on MedPredict reminded me why I love interdisciplinary work. It combined AI engineering medicine and education in one project. Instead of trying to replace doctors or teachers with AI I wanted to build something that helps them. I believe tools like MedPredict are just the beginning of how AI can play a meaningful and ethical role in learning and healthcare.

Overall this project gave me the confidence and skills to tackle real-world problems with a thoughtful and user-centered mindset. It was a challenge but one that truly shaped the way I think about technology and its purpose.

References

1. Bartolo, M., Roberts, A., Welbl, J., Riedel, S., & Stenetorp, P. (2020). Beat the AI: Investigating adversarial human annotation for reading comprehension. Transactions of the Association for Computational Linguistics, 8, 662–678. https://doi.org/10.1162/tacl_a_00338
2. Dasigi, P., Lo, K., Beltagy, I., Cohan, A., Smith, N. A., & Gardner, M. (2021). A dataset of information-seeking questions and answers anchored in research papers. Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 4591–4601. https://doi.org/10.18653/v1/2021.naacl-main.365
3. Ganapathy, N., Kumar, R., & Bansal, A. (2024). A multimodal approach for endoscopic VCE image classification using Biomed CLIP–PubMedBERT. arXiv:2410.19944. https://doi.org/10.48550/arXiv.2410.19944
4. Gupta, A., & Waldron, A. (2023, April 14). A responsible path to generative AI in healthcare. Google Cloud Blog. https://cloud.google.com/blog/topics/healthcare-life-sciences/sharing-google-med-palm-2-medical-large-language-model
5. Helland, E. D. (2023). Tackling lower-resource language challenges: A comparative study of Norwegian pre-trained BERT models and traditional approaches for football article paragraph classification [Technical report]. https://core.ac.uk/download/578702587.pdf
6. Hu, Y. (2024). Performance exploration of Generative Pre-trained Transformer-2 for lyrics generation. Applied and Computational Engineering. https://doi.org/10.54254/2755-2721/48/20241154
7. Kumar, A., & Nandhini, M. (2025). Social media application using ReactJS. International Journal of Scientific Research in Engineering and Management. https://doi.org/10.55041/ijsrem42528
8. Kwiatkowski, T., Palomaki, J., Redfield, O., et al. (2019). Natural questions: A benchmark for question answering research. Transactions of the Association

for Computational Linguistics, 7, 453–466. https://doi.org/10.1162/tacl_a_00276

9. Lee, J., Yoon, W., Kim, S., et al. (2020). BioBERT: A pre-trained biomedical language representation model for biomedical text mining. Bioinformatics, 36(4), 1234–1240. https://doi.org/10.1093/bioinformatics/btz682
10. Li, H., Vasardani, M., Tomko, M., & Baldwin, T. (2022). MultiSpanQA: A dataset for multi-span question answering. In Proceedings of NAACL 2022 (pp. 684–693). https://doi.org/10.18653/v1/2022.naacl-main.90
11. Pal, A., Umapathi, L. K., & Sankarasubbu, M. (2022). MedMCQA: A large-scale multi-subject multi-choice dataset for medical domain question answering. In Proceedings of CHIL 2022. https://doi.org/10.48550/arXiv.2203.14371
12. Park, J., Johnson, S., & Pruinelli, L. (2025). Optimizing pain management in breast cancer care: Utilizing 'All of Us' data and deep learning to identify patients at elevated risk for chronic pain. Journal of Nursing Scholarship, 57(1), 95–104.
13. Pascual, D., Luck, S., & Wattenhofer, R. (2021). Towards BERT-based automatic ICD coding: Limitations and opportunities. In BioNLP 2021. https://doi.org/10.48550/arXiv.2104.06709
14. Patel, D., Raut, G., Zimlichman, E., et al. (2024). Evaluating prompt engineering on GPT-3.5's performance in USMLE-style medical calculations and clinical scenarios generated by GPT-4. Scientific Reports, 14, 17341. https://doi.org/10.1038/s41598-024-47341-5
15. Peng, Y., Yan, S., & Lu, Z. (2019). Transfer learning in biomedical natural language processing: An evaluation of BERT and ELMo on ten benchmarking datasets. In Proceedings of the 18th BioNLP Workshop (pp. 58–65). https://doi.org/10.18653/v1/W19-5007
16. Rasmy, L., Xiang, Y., Xie, Z., Tao, C., & Zhi, D. (2021). Med-BERT: Pre-trained contextualized embeddings on large-scale structured electronic health records for disease prediction. NPJ Digital Medicine, 4, 86. https://doi.org/10.1038/s41746-021-00455-y
17. Solomou, C. (2023). Enhancing medical specialty assignment to patients using

NLP techniques. arXiv:2312.05585. https://doi.org/10.48550/arXiv.2312.05585

18. Stankevičius, L. (2024). Extracting sentence embeddings from pretrained transformer models. Applied Sciences, 14(19), 8887. https://doi.org/10.3390/app14198887
19. Su, P., Peng, Y., & Vijay-Shanker, K. (2021). Improving BERT model using contrastive learning for biomedical relation extraction. In Proceedings of the 20th Workshop on Biomedical Language Processing (pp. 1–10). https://doi.org/10.18653/v1/2021.bionlp-1.1
20. Tinn, R., Cheng, H., Gu, Y., et al. (2023a). Fine-tuning large neural language models for biomedical natural language processing. Patterns, 4(8), 100729. https://doi.org/10.1016/j.patter.2023.100729
21. Tinn, R., Cheng, H., Gu, Y., Usuyama, N., Liu, X., Naumann, T., Gao, J., & Poon, H. (2023b). Fine-tuning large neural language models for biomedical natural language processing. arXiv:2112.07869. https://doi.org/10.48550/arXiv.2112.07869
22. Wang, X., Zhang, Y., Ren, X., & Zhang, J. (2019). Fine-tuning BERT for biomedical named entity recognition: An empirical study. JMIR Medical Informatics, 7(3), e14830. https://doi.org/10.2196/14830
23. Zhang, X., Liu, Y., Wang, C., & Li, Z. (2024). Comparison of prompt engineering and fine-tuning strategies in large language models in the classification of clinical notes. Journal of the American Medical Informatics Association, 31(2), 300–312. https://doi.org/10.1093/jamia/ocad249

Appendix A

A.1 Limitations of General-Purpose Transformers in Clinical QA

Despite the promise shown by domain-specific models like PubMedBERT, several studies have underlined the limitations of adapting transformers to the complexity of clinical text. Tinn et al. (2023)**[20]** reported that large language models (LLMs) often suffer from domain mismatches, low sensitivity to clinical negations, and difficulties in interpreting long contextual dependencies. These challenges are especially significant in MCQA tasks that demand factual recall, clinical reasoning, and multi-step deduction.

Similarly, Pascual et al. (2021)**[21]** studied BERT-based models for automatic ICD coding from clinical notes and observed issues such as label imbalance and contextual

ambiguity—concerns also found in MCQA datasets where distractor choices are semantically close. These insights influenced the design of MedPredict, particularly the emphasis on confidence-aware outputs and the use of curated data subsets for evaluation.

A.2 Biomedical Relation Extraction and Contrastive Fine-Tuning

Another important area of research is contrastive learning and relation extraction within biomedical NLP. Su et al. (2021)**[19]** proposed a contrastive approach to enhance BERT's performance in relation extraction by leveraging semantic distance and contextual contrast, thereby improving generalization. While MedPredict does not yet incorporate contrastive fine-tuning, its principles offer a promising avenue for handling subtle distractors and tagging question difficulty.

Additionally, Wang et al. (2019)**[22]** demonstrated that BERT's bidirectional encoding offers substantial advantages in biomedical NER tasks, particularly in identifying contextual entities—skills directly relevant for parsing MCQ stems and linking them to potential answer options.

A.3 Structured Data Embeddings and Med-BERT

Unlike most biomedical transformers trained on unstructured text, Med-BERT (Rasmy et al., 2021)[16] was designed to process structured electronic health record (EHR) data for downstream tasks such as disease prediction. Although the nature of EHR data differs from MCQ inputs, its use of structured label sequences is conceptually aligned with MCQA's classification format. Med-BERT's emphasis on temporal and contextual embeddings presents future possibilities for hybrid systems combining structured clinical data with language-based inference.

A.4 Transfer Learning and Pretraining Comparisons

Peng et al. (2019)[15] conducted a broad comparison of pretraining strategies across ten biomedical datasets, concluding that domain-specific pretraining consistently leads to improved performance. This is especially true when labeled data is scarce, a common scenario in specialized medical domains. Their findings support the decision to adopt PubMedBERT in MedPredict and reinforce the value of transfer learning for MCQA tasks.

A.5 Prompt Engineering vs. Fine-Tuning in Clinical Tasks

Prompt engineering has recently emerged as an alternative to traditional fine-tuning, particularly in large-scale generative models. Zhang et al. (2024)[23] compared both methods for classifying clinical notes and found that while prompting offers flexibility with fewer parameters, fine-tuning remains more accurate and stable—especially for tasks with fixed output formats like MCQA. This informed MedPredict's methodology: supervised fine-tuning of PubMedBERT, rather than prompt-based learning, was chosen for its stronger alignment with classification-based QA tasks and easier integration of confidence estimation.

A.6 Multimodal Models and Future Potential

Recent biomedical NLP work explores multimodal models that combine text with other data types, such as images or signals. Ganapathy et al. (2024)[3] introduced Biomed CLIP-PubMedBERT for endoscopic image classification, demonstrating the adaptability of language models like PubMedBERT in multimodal settings. While not directly relevant to MCQA, their approach opens up opportunities for MedPredict to evolve toward diagram-based question answering or visual clinical simulations.

Similarly, Solomou (2023)[17] developed NLP tools for specialty assignment from

clinical referrals. This aligns conceptually with MedPredict's symptom checker, which applies fuzzy logic to suggest differential diagnoses—further highlighting the overlap between clinical triage, education, and language-based tools.

A.7 The Role of MedMCQA: A Benchmark for Medical MCQA

MedMCQA serves as the primary dataset and benchmark for MedPredict. It comprises over 194,000 MCQs from high-stakes Indian exams (AIIMS, NEET-PG), spanning 21 medical subjects. Its scale, diversity, and inclusion of distractor options (A–D) make it ideal for multiple-choice classification tasks.

Compared to datasets like PubMedQA and MedQA-USMLE, MedMCQA is both more varied and more challenging. The baseline PubMedBERT model achieved ~40% accuracy, while MedPredict's fine-tuned version improved this to 46.33% on the full test set These gains reflect the benefits of task-specific fine-tuning and dataset cleaning.

Moreover, MedMCQA's format allows MedPredict to offer real-time prediction, confidence visualization, and difficulty scoring. These capabilities are difficult to implement with generative datasets, making MedMCQA uniquely suited to interactive student tools.

A.8 The Gap in Literature: Student-Facing Biomedical QA Tools

Despite the proliferation of medical LLMs and benchmarks, few tools are designed for student use. Research platforms such as Med-PaLM or BioBERT showcase strong accuracy but lack interactivity, explainability, and accessibility for learners. MedPredict directly addresses this gap. It transforms fine-tuned biomedical LLMs into interactive educational companions. The system not only provides answers but also visualizes confidence scores, simulates clinical cases, and allows real-time feedback all within a student-friendly web interface. This literature review supports the foundations of MedPredict, namely:

- That domain-specific pre training improves model alignment with medical language,
- That fine-tuning on labeled MCQA data enhances task accuracy,
- And that user-centered design is essential for real-world impact in education.

MedPredict expands upon this foundation by implementing a full-stack, modular system that promotes transparency and student engagement. While it does not aim to

surpass models like GPT-4 in raw performance, it provides a replicable blueprint for how smaller, open-source models can be deployed in meaningful, high-impact educational applications.

Statement

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得四川大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

I hereby declare that the academic thesis that I submit is the research work and results that I conducted and achieved under the guidance of my adviser . To my knowledge, except where especially noted and acknowledged, this thesis doesn't contain any research findings that have been published or written by others, or any materials that have been used to obtain degrees or certificates from Sichuan University or other Educational Institutions. Any contribution made by the comrades working with me to this research has been clearly illustrated and acknowledged in this thesis.

本学位论文成果是本人在四川大学读书期间在导师指导下取得的，论文成果归四川大学所有，特此声明。

I also hereby declare that the results of this academic thesis are achieved under the guidance of my adviser, during my study in Sichuan University and they are owned by Sichuan University.

Thesis Author (Signature)

Thesis Advisor (Signature)

2025年 05月 20日

Acknowledgement

I would like to express my deepest gratitude to all those who supported and guided me throughout the completion of this thesis.

First and foremost, I extend my heartfelt thanks to my adviser Professor Tang Mingjier for his unwavering support, insightful guidance, and professional expertise throughout the course of this research. His patience, encouragement, and invaluable feedback helped shape this project from its inception to its final form. I am especially grateful for the academic freedom he granted me, which allowed me to explore ideas independently while always having his mentorship as a steady compass.

I am also thankful to the faculty members and staff of the College of Software Engineering at Sichuan University. Their teaching and mentorship over the past years provided me with a solid foundation in both theory and practice, which proved essential for this research project. I would also like to acknowledge the resources and infrastructure provided by the university, including access to computing facilities and online platforms.

Special thanks go to the students and all participants who engaged with the MedPredict platform and provided constructive feedback during testing. Their enthusiasm and input played a vital role in evaluating the usability and effectiveness of the system.

I am deeply indebted to my family, particularly my parents Mhamed Filali & Hanane Makani and brother Nizar Filali, for their unconditional love, moral support, and faith in my journey. Their sacrifices, encouragement, and emotional presence were my greatest source of strength. Without their constant belief in me, this academic milestone would not have been possible.

Lastly, I thank everyone who, in one way or another, contributed to the success of this project. While it is not possible to name everyone individually, please know that your support has been appreciated.

This thesis represents not only the culmination of my undergraduate journey but also the collective contributions of many individuals to whom I am sincerely

grateful.